Opponent-based Tactic Selection for a First Person Shooter Game

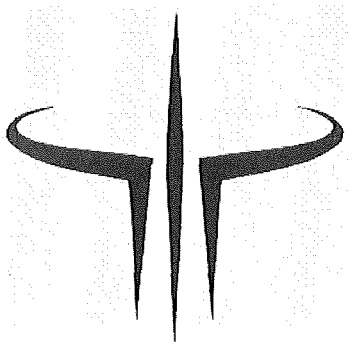
A thesis submitted in partial fulfillment of the requirements for the Degree of

Master of Science in Computer Science and Software Engineering, by

David Thomson

Supervisor: Tanja Mitrovic

University of Canterbury, 2010



Acknowledgments

I wish to deeply thank my supervisor Tanja Mitrovic for letting me do something so much fun.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

For better or for worse, video games are becoming a significant part of our society. While once little more than a curiosity, the video game industry has grown rapidly, to the point where it generated about USD$9.5 billion in the US in 2007, and 11.7 billion in 2008 [1].

Modern personal computers (PCs) owe many advancements to the gaming industry: sound cards, graphics cards, 3D acceleration, and CD/DVD drives, among others. Sound cards were developed for adding sound to computer games, and were only later used for music production. Graphics cards were needed for more colors in games, which then allowed for graphical user interfaces (GUIs). CD and DVD drives were developed for mass distribution of media, including games. Modern games are some of the most demanding applications in terms of PC resources, and push the speeds of CPUs up, thus reduce the price of commodity CPUs.

As well as hardware, video games have contributed to many advances in software. Perhaps most notably, Unix itself was developed partly so the programmers could play a space game [2]. Video games often employ complex Artificial Intelligence (AI) techniques, and can be very useful environments for testing different AI methods.

User modeling is a general term used for collecting and processing data about each user [15]. This information could be preferences, habits, or knowledge. The information is usually stored and used at a later date, to tailor the system by selecting material or strategies best suited for a particular user. Sometimes, the purpose of user modeling is simply to display such information back to the user to promote self-reflection [8]. Systems that implement user modeling can adapt to each user and thus provide a more enjoyable or useful experience to different types of users. Difficulties in user modeling arise when deciding what information to record, and how to actually use this information. There is little point in recording information the system cannot use, but it is impossible to use information that cannot be recorded. Systems that employ user modeling techniques range from games, to educational systems (discussed below), and even search engines [19]. A very common form of user modeling is providing recommendations, as seen on the webstore, amazon.com. During previous visits, AMAZON learns about a users interests, and builds a model of products positively valued by the user. The site can then use this model to recommend other products that may interest the user, potentially leading to more sales [22].

A common application of user modeling is found in Intelligent Tutoring Systems (ITSs), called student modeling. ITSs attempt to build a model of the student's knowledge, and use this model to tailor the instruction. Recommending problems of an appropriate difficulty level is a common use of the model, as is recommending certain topics the student should study. Student modeling has been shown to be an effective way to help students learn, and is used by many, if not all Intelligent Tutoring Systems [11, 17, 18].

User modeling has been incorporated into some games, under the term Opponent modeling. Some such games include Poker [4], Scrabble [5], Real-Time Strategy Games [6], and Racing Games [7]. Results vary, but generally the authors conclude that opponent modeling has significant potential, but is hard to perfect. Because the possible information to record is different in every game, any opponent modeling implementation is usually very game-specific. These systems are discussed in detail later in this report. As graphics and physics engines approach the limit (e.g. 100% realistic), more emphasis will be placed on the Intelligence of the game. An adaptive game that always challenges the player and can match strategies employed by the player may ultimately decide the success of the game, if this is not already the case. Opponent modeling has the potential to make games more interesting, and to remain interesting for a longer time [10].

This project aims to implement opponent modeling in a similar way to how an ITS models students, to make a game system more proficient at defeating a user, and adapt to strategies a user might employ. Such adaption

should motivate the players to continue playing the game, as it will continue to provide a challenge. Motivation is important, whether in an Intelligent Tutoring System, where students need motivation to learn, or in a commercial game, where the game needs to have long lasting appeal to be successful. Our research aims to implement opponent modeling to help motivate players to continue playing, by providing a lasting challenge. Motivation is also a very important factor in educational games, so techniques for improving motivation could be beneficial to a number of fields.

We start by describing Student Modeling, some previous applications of Opponent Modeling, and Quake 3, the game which is the context of our work. Section 3 then describes our implementation of opponent modeling in Quake 3. Section 4 gives the results of evaluation, and Section 5 concludes with discussion and future work.

Chapter 2

Background

2.1 Quake 3

Quake 3 is a death-match style First Person Shooter (FPS) game. FPS is a video game genre which centers the game-play around gun- and projectile weapon-based combat through the first person perspective. The player sees the action through the eyes of the player character, and the primary design element is combat, almost exclusively involving firearms. Wolfenstein 3D, released in 1992, and Doom, in 1993 popularised the genre, which as of 2006, in terms of revenue for publishers, was one of the biggest and fastest growing video game genres [16]. On December 2nd 1999, *id software* released Quake 3 Arena, the third installment of the popular Quake series. Quake 3 Arena, also known as Quake 3, Q3A or Q3, focuses on multi-player action, and has no story line. In single player mode, the player competes in a series of

matches against computer-controlled characters, with the same game-play as would be found in a multi-player game against other human players. The most common form of play is the *death-match*. In a death-match, all players compete against each other, on a single map. A map is a pre-constructed area where opponents battle in a match. A map contains numerous areas, with rooms, corridors, and openings. Some maps feature dangerous obstacles, such as lava, that kills a player who is unlucky enough to fall into them. Some maps are set in space, where the player must be careful not to fall off the edge into oblivion. Maps are designed for a certain number of players, although players can easily be added or removed before the game begins. Power-up items that can be picked up are distributed throughout each map at pre-defined locations. These power-ups include health kits, weapons, and ammunition. The winner of a match is the player who scores the most points, achieved by killing other players, in the match time limit. Players can also achieve victory by reaching a points limit (if there is one) before any other player. As well as a *death-match*, there are other forms of play including *team death-match*, where players compete in a team to score more kills than the other team, and *Capture The Flag* (CTF), where players must reach their opponents base, capture the flag, and return it to their own team's base.

Figure 2.1: A screenshot from Quake 3

Figure 2.1 shows a screenshot from a Quake 3 match. In the bottom center is the score representing the characters health, in this case 38. When a character shoots at and hits an opponent, the opponent loses some of its health. How much it looses depends on the weapon that was used. If this causes the character's health to reduce below zero, the character dies. After a brief pause, the character re-spawns (comes back to life) at a different location on the map, and with only basic weapons. This ensures players

continuous game-play, even if they get killed a lot. There are a number of different weapons in the game that can be found scattered around a map. A player can pick up such weapons and use them against their enemies. A weapon is useless without ammo, so the remaining ammo for the current weapon is displayed on the bottom left of the screen, as shown in Figure 1. In this case the player has 100 ammo remaining. Three scores are shown in the right of Figure 2.1: The score limit (40), the players score (0), and the score of the best opponent (also 0 at this time).

On August 20 2005 the complete source code for the Quake 3 engine was released under the open source GPL. From this formed the ioquake3 project, which aims to improve the original source release with bug fixes and updates. Improvements include new audio systems, VoIP, IPv6 support, various ports to new platforms and architectures, and security fixes. This project uses version 1.36_SVN1754M of the ioquake3 project.

2.1.1 Quake 3 Artificial Intelligence

Quake 3 uses a wide range of techniques and common sense solutions to control artificial players, known as bots. Bots allow everyone to enjoy the game without requiring a network connection to other people, and can also be useful for training purposes. The bot is an artificial player that "lives" inside the computer, and receives information about the game environment directly from the game itself as a set of variables. The bot then acts on these variables in a way designed to appear as human-like as possible. The bots in

Quake 3 were specifically designed to act as and be hard to distinguish from a human player. This means the behaviour must not be pre-defined, and the bot must react to new situations in a reasonable manner. The Quake 3 bot has a complex navigation system which allows bots to navigate through new environments (maps) effectively and a relatively simple cognitive model makes the bot quite resource efficient. The Quake 3 bot is summarised in this section; for a complete description see [14].

2.1.1.1 Bot Characters

A human player can play a game of Quake 3 with one or more artificial players called bots. To make the game more enjoyable, a range of bot characters exist that play the game in their own style, and present different challenges to a human player. There is a set of characteristics that influence how a bot plays the game and these characteristics are varied for each bot. The characteristics, as shown in Figure 2.2, are all related to the game; things such as hair style etc. are not of interest (although there is of course a different model for each bot).

The characteristics have been carefully selected to minimise contradiction between them, and are all normalised. It is important to note that the most important aspect is the perception of human players. What human players think about how a bot plays the game is more important than how the bot actually plays the game. The characteristics are all predefined, and set, for each bot. Setting the difficulty changes some of the characteristics for each

bot. When the difficulty is changed, a new set of pre-defined characteristics is loaded, for the particular difficulty level. Many characteristics stay the same at different difficulty levels, but some, that influence the characters effectiveness, namely Attack skill and Aim skill, are changed.

Name	Name of the bot.
Gender	Gender of the bot (make, female, it - mechanical creature).
Attack skill	How skilled the bot is when attacking. > 0.0 & < 0.2 = don't move $>= 0.2$ & < 0.4 = only move forward/backward $>= 0.4$ & < 1.0 = circle strafing > 0.7 & < 1.0 = random strafe direction change > 0.3 & < 1.0 = aim at enemy during retreat
Weapon weights	File with weapon selection fuzzy logic.
View factor	Scale factor for difference between current and ideal view angle to view angle change.
View max change	Maximum view angle change per second.
Reaction time	Reaction time in seconds.
Aim accuracy	Accuracy when aiming, a value between 0 and 1 for each weapon.
Aim skill	Skill when aiming, a value between 0 and 1 for each weapon. > 0.0 & < 0.9 = aim is affected by enemy movement > 0.4 & $<= 0.8$ = enemy linear leading > 0.8 & $<= 1.0$ = enemy exact movement leading > 0.6 & $<= 1.0$ = splash damage by shooting nearby geometry > 0.5 & $<= 1.0$ = prediction shots when enemy is not visible
Chats	File with individual bot chatter.
Characters per minute	How fast the bot types.
Chat tendencies	Tendencies to use specific chats when things happen.
Croucher	Tendency to crouch.
Jumper	Tendency to jump.
Walker	Tendency to walk instead of run.
Weapon jumper	Tendency to rocket jump.
Item weights	File with item goal selection fuzzy logic.
Aggression	Aggression of the bot.
Self preservation	Self preservation of the bot.
Vengefulness	How likely the bot is to take revenge.
Camper	Tendency to camp.
Easy fragger	Tendency to go for cheap kills.
Alertness	How alert the bot is.
Fire throttle	Tendency to fire continuously instead of pausing between shots.

Figure 2.2: The characteristics that define a Quake 3 bot

Most of the characteristics are in the range [0, 1]. The higher the value the more the characteristic is true. For example, if a bot has the *camper* characteristic of almost one, the bot will camp almost all of the time. The characteristics *Weapon weights* and *Item weights* refer to locations where fuzzy logic is stored for situation dependent weapon preferences, and item goal selection respectively. Bots are seen most often when fighting, so most characteristics are concerned with the fighting behaviour of the bot. Other characteristics, such as Chats, Characters per minute, and Chat tendencies define how the bot talks to other players. In Quake 3, players can type messages to other players. Continuing the theme of acting like human players, the bots in Quake 3 are rather talkative. Each bot says different, often humorous things in response to events in the game. The chatting nature of the bots is not further explored in this thesis.

Changing the characteristics of a specific bot can lead to interesting behaviour. A perfect bot could be easily made, but this would not be very enjoyable for a human player. A human player needs to win at least some of the time, or at least know that with practice they will be able to defeat the bots. A bot that is just a little better than the human player can be suitable for training and practice, as well as providing an exciting challenge.

The impact of slightly changing characteristics can sometimes go unnoticed. At the same time, some changes can drastically change the bot's behaviour. The bots in Quake 3 all have characteristics that have be finely tuned to result in bots with a wide variety of different playing styles, and

that are enjoyable opponents for a human player.

2.1.1.2 Bot Decisions & Preferences

The bots in Quake 3 use fuzzy relations to specify how much the they want to do, have, or use something. A fuzzy value is attached to each item the bot might want, and is based on the current state of the world and the state of the bot. For each item, the bot has a fuzzy relation which shows how much the bot wants to have the item. The values for each item can be effected by items the bot has in its possession, how much of the item the bot has, and how much health and armor the bot has. The bot evaluates the fuzzy relation for each item, and desires the item with the highest weight. In the same way the bot has a fuzzy relation for each weapon to choose the right weapon during combat.

The fuzzy relations that the bot uses are stored using a tree-like structure. The leaves of the tree store the value; a node in the tree links to either a leaf or another node in the next level of the tree. The fuzzy relations are stored in plain text using a C-like syntax.

```
weight "name"
{
 switch( /*one of the criteria*/ )
 {
  case /*smaller than a certain value*/:
  {
   switch( /*one of the criteria*/ )
   {
    case /*smaller than a certain value*/: return /*a fuzzy value*/;
    case /*smaller than a certain value*/: return /*a fuzzy value*/;
    default: return /*a fuzzy value*/;
   }
  }
  case /*smaller than a certain value*/: return /*a fuzzy value*/;
  case /*smaller than a certain value*/: return /*a fuzzy value*/;
  default: return /*a fuzzy value*/;
 }
}
```

Figure 2.3: Fuzzy relation structure

Figure 2.3, which originally appears in [14], shows the structure of the fuzzy relations used by the Quake 3 bot. Every fuzzy relation (called a weight) has a unique name that identifies it. C-like switch statements represent the nodes of the tree-like structure. The switch statement selects one of the criteria that represents some part of the state of the world. Case statements then divide the value of the criteria into ranges. A switch statement always has a default statement, that occurs when none of the case ranges apply. A leaf of the tree denoted by the keyword *return* stores a fuzzy value. This tree structure is easy to understand and modify. All bots have the same tree-structure, but different weight values.

The bot uses weights to decide which item it wants most, and also which weapon to use during combat. An example of each case, is presented below. For a complete description, see [14].

```
weight "holdable_teleporter"
{
 switch(INVENTORY_TELEPORTER)
 {
  case 1:
  {
   switch(INVENTORY_MEDKIT)
   {
    case 1: return 60;
    default: return 0;
   }
  }
  default: return 0;
 }
}
```

Figure 2.4: The holdable_teleporter fuzzy relation

Figure 2.4 gives the fuzzy relation for the teleporter item, which is a holdable item. INVENTORY_TELEPORTER is a variable that is 1 when the player has a teleporter. INVENTORY_MEDKIT is a variable that is set to 1 when the player has a med kit. A player can only hold one holdable item at any time. A fuzzy weight higher than zero is therefore only returned when the bot has no teleporter and no med kit. The 'case 1' case is true when the player does not have the item in the switch statement, possibly contrary to initial expectations. For each item the bot has such a fuzzy relation. The bot prefers the item with the highest weight.

```
weight "Lightning Gun"
{
 switch(INVENTORY_LIGHTNING)
 {
  case 1: return 0;
  default:
  {
   switch(ENEMY_HORIZONTAL_DIST)
   {
    case 768:
    {
     switch(INVENTORY_LIGHTNINGAMMO)
     {
      case 1: return 0;
      case 50: return 70;
      case 100: return 77;
      case 200: return 80;
      default: return 80;
     }
    }
    case 800: return 0;
    default: return 0;
   }
  }
 }
}
```

Figure 2.5: The Lightning Gun fuzzy relation

Figure 2.5 gives the fuzzy relation for the preference a bot has for using the lightning gun during a fight. INVENTORY_LIGHTNING is set to 1 when the bot has the lightning gun. ENEMY_HORIZONTAL_DIST is a variable that represents the distance towards the enemy. INVENTORY_LIGHTNINGAMMO is a variable that represents the amount of ammunition the bot has for the

lightning gun. The lightning gun is only used when both the bot has the weapon and the enemy is within range, which for the lightning gun is 768 units. The bot will prefer the weapon more if they have more ammo. If the bot has the weapon, has some ammo, and the opponent is within range, this relation will return a non-zero weight, representing the preference the bot has for using the lightning gun. This weight is then compared against the weights for all the other weapons, and the bot selects the weapon with the highest weight to use in combat. At any given time the bot can hold many weapons to choose from.

2.1.1.3 Bot Goals

The most important goal for a bot, and also a human player, is to win the match. To achieve this goal, the bot uses a number of short and long term goals. In deathmatch mode, winning is achieved by having the highest frag count (number of killed opponents) at the end of the game. The game ends when a player reaches the frag limit or when the time limit is hit, so the bot will aim to have the highest frag count when this occurs. Team death-match is won by being on the team with the highest combined frag count, while in capture the flag games the team with the most flag captures wins.

During a game the bot will use many sub-goals to attempt to win. In deathmatch mode, and when fighting an opponent in team games, the bot will try to kill an enemy it sees. To do this, the bot must aim and shoot at the opponent. The bot will also try to stay alive during battles, which involves

dodging enemy projectiles. During a fight and also when not fighting, the bot will try to pick up items of benefit, such as heath or armor, and new weapons and ammo. A bot may also decide it is going to lose a fight, and attempt to flee.

There are different types of goals the bot uses to eventually win a match. A distinction is made between short-term goals and long-term goals. Short-term goals are goals the bot achieves while working towards a long-term goal. For example, the bot picks up an item, which is a short-term goal, while chasing an opponent, which is a long-term goal. All bots have the long-term goal of winning the game, as well as others that it achieves while pursing that main goal. The bot continuously evaluates the status on achieving the current goals. If the bot wants to pick up an item, but when close sees it is not there (has been recently picked up by another player), the bot will abandon the goal. The bot also keeps track of the time it is taking to achieve a goal, and at some point may decide it is taking too long and abandon the particular goal.

2.1.1.4 Bot Fighting

Bots are most often seen during direct fights, so the observed behaviour during a fight is very important. A lot of settings of bot characters apply to fighting behaviour. Before the bot can fight it needs to acquire an enemy. If the bot gets attacked it will quickly look around in the direction of the attack and search for the enemy. While not being attacked, the bot is continuously

looking out for danger. Information about the positions of all the players is available, but such information is not available to a human player. To increase realism, the bots field of vision is explicitly limited to 90 degrees, the same of a human player. This means the bot will not automatically see an opponent behind them, but actually has to turn around first. Environmental effects such as fog can also effect the bots ability to spot an enemy. When the bot sees an enemy, it will usually initiate a fight, if the enemy has not already done so.

Sometimes however, it is not a good idea to initiate a fight. If the bot is low on health or does not have any powerful weapons it may decide to not engage an enemy when it sees one. It might instead attempt to stay out of the opponents field of vision, and continue with its goals of picking up items, until it feels more confident.

How well the bot aims at an opponent is determined by the bots aim accuracy and aim skill characteristics. When aiming, the bot selects a spot on or near the enemy to aim at. The accuracy with which the bot chooses this spot depends on the aim accuracy skill. Depending on the aim accuracy characteristic the bot will be more or less skilled in selecting a good spot to aim at. When using an instant kill weapon. such as the Railgun, the bot always selects a spot right on the enemy, and the aim accuracy characteristic is not used. However, many weapons fire projectiles that travel at a slower speed. For such weapons, aiming at where an opponent is will not work, as the opponent will most likely not be there by the time the projectile

reaches the spot. Instead, the bot must aim at a spot somewhere in front of the opponent. The aim skill characteristic determines how well the bot can predict where the opponent will be when the projectile reaches them. Bots with a high aim skill also take into account the surrounding geometry. Some weapons do splash damage, so hitting a nearby geometry may damage the opponent without needed a direct hit. When the enemy goes out of sight the bot will attempt to predict where the opponent will re-emerge, and will shoot projectiles that inflict splash damage at that location.

This is only a small description of the aspects of the Quake 3 game that are relevant to this research. For a complete description, please refer to [14].

2.2 Applications of Opponent modeling

2.2.1 Poker

Billings *et al.* [4] describe and evaluates a poker program capable of "*observing its opponents, constructing opponent models and dynamically adapting its play to best exploit patterns in the opponents play*". Poker is an interesting game because it is a game of imperfect knowledge. The computer can not simply compute all possibilities and select the best option, as in chess or other board games. A very important aspect of poker is studying the opponent, and trying to guess the purpose of their actions. Poker, or more specifically Texas Hold'em poker is therefore a game where opponent modeling could be utilised to significantly increase the skill at which the computer

can play. The applied technique combines assigning a generic weight to each players hand that represents the possibility of the opponent having a good hand with a specific set of weights for each opponent, based on their betting history. Each time the opponent makes a betting action, the weights for that opponent are modified. A raise by the opponent will increase the weights for the strongest hand given the flop cards and decrease the weights for weaker hands. The weights are then used to calculate probabilities about each opponents hands, and actions are made accordingly. Simulations were conducted where different versions of the computer competed against each other. Five different AI implementations where evaluated (Basic Player Modeling, Basic Player Tight, Basic Player Loose, Generic Opponent Modeling and Specific Opponent Modeling). The results showed both opponent modeling implementations to be very effective at defeating the other computer players. Generic Opponent Modeling (no betting history information is used) was better than previous techniques, but Specific Opponent Modeling, where the betting history of the opponent is used, was shown to be much more effective. The authors also evaluated the system online against human players, where it consistently won, although not enough reliable data was obtained to say that the opponent modeling versions outperformed the previous best program in these games. The authors note that Specific Opponent Modeling was hampered by the crude methods used for collecting and applying observed statistics.

2.2.2 Scrabble

Richards and Amir [5] describe an opponent modeling approach to scrabble. Scrabble is a stochastic partially observable game. Like poker, players have incomplete knowledge on the game state. Although games of imperfect knowledge can be modeled formally by partially observable Markov decision processes[9], solving them is intractable for problems containing more than a few states. The authors hypothesise that opponent modeling could be used to help overcome the intractability problem, while still giving a significant improvement in playing ability. The opponent modeling technique described by the authors differs from other methods as it is more concerned with guessing what the opponent currently has. This is done mainly by probabilistic methods, taking into account the move the opponent just played. An important aspect of scrabble game play they note is it is crucial to consider what letters a move leaves you with, as well as the points a play will immediately score. This point is used to try to guess what letters the computer has, and therefore what the optimal next move is. Although computers are already very good at scrabble due to perfect knowledge of all possible words, this system achieves a significant improvement on the current best technique. All the evaluations were done between two computer players, so results may vary with human players. The authors note that they do not expect opponent modeling to be useful in all situations.

2.2.3 Real-Time Strategy Games

Schadd *et al.* [6] describe an implementation of opponent modeling into a Real-Time Strategy (RTS) game. RTS games provide an interesting environment, where the AI is expected to utilise complex and realistic strategies against opponents. The AI in RTS games must perform in real time (hence the name) which limits the amount of computation that can be performed. At the same time the computer must also handle the graphics, physics etc. RTS games are simulated wargames, where players must build bases and units to attack and defeat the enemy. Each unit has different strengths and weaknesses, and a good player will build units and adopt a strategy that plays at an opponents particular weaknesses. RTS are a challenging but potentially very rewarding application for opponent modeling. The approach taken by the authors is to first classify some generic models that cover the particular strategies a player might employ. The models are employed in a hierarchy consisting of two levels. The top level is most abstract and classifies a player into either an offensive or defensive strategy. The second level classifies a player based on what units the player builds. During the game, the AI system attempts to classify the player into one of these categories. The best strategy against that particular category is then deployed against the opponent. The main difficulty is in gathering sufficient data to be confident about the strategy of the opponent. Once this is decided, selecting the best strategy is simple as there is a finite number of categories the opponent can be classified into. Experimental results show that the game can suc-

cessfully classify opponents, although some time is needed to before accurate classifications in both levels are achieved.

2.2.4 Racing Games

Togelius *et al.* [7] attempts to implement opponent modeling in a car racing game. In most car racing games the goal is to drive around a track as fast as possible. Some games involve racing against other players, while in others the player races against the clock, trying to beat a certain time. Although racing games have evolved considerably in terms of graphics and physics, little progress seems to be being made on improving the AI. The authors have two distinct motivations: to *"create more adaptable, believable and entertaining games through using AI"*, and to *"explore the suitability of video games as environments for studying the emergence of general intelligence"*. Initial attempts at opponent modeling have limited success. A human player drove around a track, and inputs from sensors on the car and the driver's actions were recorded. This data was then used to train a neural network, to re-create the driving style of the human player. Backpropagation and Nearest-neighbour classification were tested, both with limited success. While capturing the actual driving style of a human player proved unsuccessful, the authors had more success when they attempted to learn the performance profile of a human player. This was a more abstract representation of the players skill, and modeled driving styles such as *reckless* and *careful*. These styles were used to generate new tracks, and to predict

the performance of certain drivers on these tracks. They found the generated tracks were drivable for a human player and appeared well designed. Evaluation as to whether these tracks are in fact enjoyable to a human player is *"something that definitely needs to be done."* Player modeling was successful to the extent that the authors could generate computer-controlled players that made the same progress (e.g. performed equally well) as the human driver, on a number of tracks, even though the actual driving style differed from human driving. The authors note that the success was probably limited by the inputs they had at their disposal.

2.2.5 BZFlag

Miles [23] investigates the use of machine learning techniques in computer games to create a computer player that adapts to its opponent's game-play. The context of the authors work is BZFlag (short for BattleZone Flag), a free, open source, cross-platform multiplayer 3D tank battle game. The basic game-play of BZFlag is to have two or more tanks trying to shoot each other. BZFlag is a modern game with strict performance requirements. The first objective of the research was to investigate whether machine learning techniques could be added to the game, without crippling the performance. After concluding this was possible, the author implemented and evaluated three machine learning techniques: Static prediction models, continuous learning and reinforcement learning. Static prediction provided performance similar to the existing computer player in many cases, but it was deemed unlikely

that tuning the algorithm or dataset would provide a sufficient increase in performance to match a human player.

Continuous learning on the static prediction models with the best performance showed generally positive results, and several combinations showed an improvement in performance over time. However, a 'plateau' effect was found, and one of the algorithms showed a decline in performance over time.

Reinforcement learning was then used to create a computer player that was able to adapt to an opponent's game-play. Initial experiments using the neural-network approach with Connectionist showed a decline in performance over time. Tuning the reward functions showed an improvement in performance, although overall performance was still poor. PIQLE was experimented with, and showed an increased rate of improvement, but much game time was needed, and performance still did not match the existing computer player.

This research showed it is possible to integrate machine learning techniques into a modern 3D game, although the results were not exemplary. Reinforcement learning showed promise, but needed a lot of game time to learn the human player's style, which limits the usefulness of any adaption.

2.3 Student modeling

Student modeling is a form of user modeling commonly employed by Intelligent Tutoring Systems. ITS track the students actions in the system, and

build a model of their knowledge [20]. This model is then used to adapt to the student. Common examples of adaption include suggesting problems and varying the level of feedback. This is done in an attempt to cater to the student as much as possible. ITS are concerned with teaching students, so the purpose of the adaption is specifically to enable the student to learn as much as possible, while not getting frustrated or bored.

Building a complete model of a student's knowledge is considered to be an intractable problem [21]. A student model should not only contain what the student knows, but also what the student does not know. It is however impossible to model everything a given person does not know. Stellan Ohlsson proposes a solution to this problem: only model what is useful, and forget about the rest [3]. There is no point trying to model information that the system cannot use. First, identify useful information the system could theoretically use, and then examine how to gather and model that information. This in part led to constraint-based student modeling. In a Constraint-based ITS, the domain is modeled by a large number of single units of declarative knowledge, known as constraints. Each constraint specifies a given truth in the particular domain. When a student submits a solution to a constraint-based tutor, their solution must not violate any of the constraints in the domain; if it does, the solution is deemed incorrect. For examples of constraint based tutors see [11]. A given problem is relevant to a subset of all the constraints in the domain, so it is theoretically possible to suggest problems that the student has not mastered. A problem relevant

to a couple of constraints that the student has recently violated may be a good candidate for recommendation. However, a problem relevant to more than a certain number of recently violated constraints might be too difficult for the student.

Following this reasoning, in a Constraint-based tutor, the student model consists of the history of each time the student has either violated or satisfied each constraint in the domain. Constraints with many violations show gaps in the students knowledge; constraints mostly satisfied show correct knowledge. The model grows over time and becomes more accurate as the student uses the system and solves more problems.

```
...
(53 0 (1) 1)
(54 0 (1 1 0 1 0 1 1) 3/5)
(60 0 (1 1 1 0 0 0 1) 2/5)
(69 0 (1 1 1 1 1 1 1) 1)
(70 0 (1 1 1 1 1 1 1) 1)
(71 0 (1 1 1 1 1 1 1) 1)
...
```

Figure 2.6: A section of a student model

Figure 2.6 shows a section of a student model from EER-Tutor [12]. The constraint number is given, followed by a list of zeroes and ones. A one represents the constraint being satisfied; a zero represents violation. A value representing the students overall skill on the particular constraint, normalised between zero and one follows the list. This is a conceptually simple but practically useful representation of the students knowledge. Superfluous data

is not modeled and the model is computationally simple to construct. Such student models have been shown to be useful when used in an ITS [11, 12, 13].

Chapter 3

Opponent Modeling in Quake

3.1 Motivation

This thesis describes an implementation of opponent modeling in Quake 3 Arena. Although the computer-controlled characters, or *bots*, as they are known, are already capable of playing the game in an effective manner, we wanted to examine whether modeling opponents would make a measurable difference to the bots skill. A lot of work has gone into making the bots in Quake 3, and many other games, challenging and entertaining to play against, but we believe an aspect of artificial intelligence has been overlooked in many computer games, namely, adaptation. Being able to adapt is a vital skill humans have, and has probably played a part in our rise to dominance. Being able to adapt to different living conditions, weather, food, and enemies have all contributed to the progress of the human race. It is our ability to

adapt, rather than our skill at any given activity, that has led to our success. In light of this, it seems that to reach the pinnacle of intelligent computers, adaption must be a key goal. The previous section of this thesis introduced one area in computer science where adaption is indeed the goal; Intelligent Tutoring Systems. Intelligent Tutoring Systems owe some of their success to being able to adapt to different students. This is an attempt to computerize the natural skill a human tutor has for adapting to different students. An important idea from research in this field is that when modeling students, the usefulness of the model is more important than its completeness. There is no point modeling information that cannot be used, and it is unnecessary to construct a complete model of the students knowledge.

Adaption has been implemented as opponent modeling in some games, as described in Section 2. Although the implementations show promise, results are often mixed, and benefits are often outweighed by added complexity. It could be possible that developers have been too wary of computational complexity to develop complex adaption engines. Most computer games are very CPU/GPU intensive, so computationally complex components with mixed benefits might not be included, or even investigated. It is much easier to market an increase in graphic effects, as potential consumers can immediately see the improvements, and are attracted to the game. The importance of having the latest and greatest graphics may have come at the cost of developing useful adaptation techniques. However, at some point we must reach a situation where the graphics are simply as good as they need to be: photo-realistic,

with perfect shadows, reflections, and other effects. At this point, developers may have to investigate other areas in which to differentiate their games. The rise of commodity priced dedicated graphics cards also means much of the work is being offloaded onto the GPU (Graphics Processing Unit), freeing up the CPU (Central Processing Unit) for other tasks. Now is the perfect time to investigate and develop robust adaptation techniques for computer games. While some work is surely occurring in industry, we believe this is an exciting field for research. Adaption in computer games is also crucial to the effectiveness of educational computer games, a field that is rapidly growing in size and importance.

3.2 Theory

In humans, the ability to adapt comes from our ability to remember. When faced with a decision, we recall, either sub-consciously or consciously, previous situations where we had to make similar decisions. We examine the decision we made on that occasion, and whether the result was desirable, or not. If our previous decision led to a desirable outcome, we are likely to make a similar decision. If, on the other hand, the previous decision led to undesirable outcomes, we are likely to make a different decision. In essence, we are remembering what worked well in the past, and what did not. An important aspect of the skill is in identifying previous situations that are relevant to the current decision. This is true in many domains, from sport to

social interactions. As we grow older, we learn which moves in a particular sport lead to success, or which behaviours lead to desirable social outcomes. What is desirable is of course dependent on the person themselves, which means that simply doing what works for other people is not sufficient: we must learn the correct actions ourselves.

To do this we must define what is desirable, and we must also have some way of identifying similar previous situations. When implementing adaption in a computer program, the former is generally easy, while the latter is more difficult, and often the cause of uninspiring results. Before developing an adaption system, it must be clear in our heads what is desirable, and how we are going to overcome the problem of computers unsuitability for identifying *similar* situations. At this point it is helpful to remember what student modeling has taught us: there is no point modeling what you cannot use, and you cannot use what you cannot model. This helps to give us a place to start. Once we know what is desirable, we can identify decisions in situations that lead to desirable outcomes. If a decision does not at all influence the outcome, there is little point modeling it. If we cannot change a decision that influences the outcome, even with more information, then there is also no point modeling that decision. We must only model decisions that we can change, and that influence the outcome, in either a desirable or undesirable manner. Modeling decisions that lead to an undesirable outcome is important because it teaches us to be wary of making the same incorrect decision again. It may not be possible to model all possible decisions, and

some outcomes may be influenced by other variables entirely. In these cases evaluation should be incorporated into the development process. Again, student modeling teaches us that a simple approach that adds value is better than a theoretically complete solution that gives no practical benefits.

3.3 Design

Applying these principles to Quake 3, we must define the desirable outcomes, and which decisions, that we can change, effect these outcomes. We will build models of the opponents in a match, consisting of decisions made, and the outcomes that occurred. We do not intend our models to be complete representations of a bots playing style, rather something useful that is not too cumbersome to construct. Performance is *still* very important for most computer games. Quake 3 was chosen because it was a commercial success, and a very popular game. The Quake 3 engine was released under an open source license, allowing for research unconstrained by restrictive agreements. Quake 3 was also considered because of the minimalist playing style: unnecessary features, modes and items do not exist, making defining and building the models simpler. Although Quake 3 is used as the context of this research, the concepts and indeed implementation should be applicable to any First Person Shooter game. Opponent modeling techniques are generally game specific, as one is building a model of decisions made in the game. This means that developers wishing to incorporate adaption into their game often

have to start from scratch, both in terms of design and implementation. This project aims to design and develop an adaption system that will work for any First Person Shooter game. A developer could implement this design, and possibly extend it for their particular game, making adaption more accessible to small-scale game developers.

For a bot in Quake 3, the obvious desirable outcome is to win the match. This is much too high level, however. Many decisions, in many situations, lead to a bot winning a match; such decisions should be modeled for *each* situation. At a lower level, the desirable outcome for a bot is to kill an opponent, when engaged in combat. As bots navigate throughout a map, they will often become involved in combat. Combat simply means the bot is currently seeing an enemy, who it is shooting at. In this combat situation, the desirable outcome is to kill the opponent. Conversely, the undesirable outcome is being killed by the opponent. There is also the possibility of neither bot being killed. This may occur if one bot flees while low on health, or perhaps a third player enters combat with one of the bots, changing the situation. This gives us our situations: combat with an enemy. If the bot wins most of these encounters, by achieving the desirable outcome, they will have a greater chance of winning the match. Our models will record the results of all combat between bots, and whether the desired outcome of killing the opponent was achieved.

Given the situation of combat, we must investigate which decisions lead to achieving the desired outcome. Obviously, the skill of the bot influences

the outcome. A bot who misses a lot is going to be less successful than a bot who scores a perfect hit every time. Changing the skill of the bot would change the outcome, but this is too simplistic. It would be easy to make a perfect bot, that always hit the enemy, but such a bot would not be fun to play against. The purpose of this research is to make the bots better, but by making them smarter, not just by changing their parameters. We therefore must look at other variables that influence the outcome of a combat situation. First Person Shooter games are primarily about, as their name suggests, shooting. For the enjoyment of the player, and the lasting appeal of the game, there is always a collection of different weapons to choose from in such games. Some players may prefer the precision and cunning of a sniper-rifle, while others may enjoy the brute force and destruction of a rocket launcher. Experienced campaigners might realise that each weapon has its moment, and selecting the right weapon for the occasion increases the chance of success.

This leads us to three decisions we wish to be able to make with help from information regarding a particular opponent. These decisions are which weapon to choose, which weapon(s) to look for, and whether to chase or flee an opponent. If the choice of weapon is influential to the outcome of a given combat situation, then by using previous situations of being engaged in combat with the same opponent, we should be able to choose a weapon that worked well for us in the past. This works well with our theory influenced by student modeling: the weapon used by a bot during combat is easy to

record, and we can easily change which weapon the bot is holding. We are only using information we can record, and we are only recording information that we can use. For every situation, which we define as combat between two bots, the bots record which weapon they chose to use, and if the outcome was desirable or not. This is obviously a simplified model of the interaction between two bots, but student modeling in Intelligent Tutoring Systems has shown us that models do not need to be complete to be useful. Such a model could be constructed in almost any First Person Shooter where the main action is combat.

Before a player can use a weapon, they must locate an item pickup on the map. Items, including weapons, always appear at the same locations; an experienced player will quickly memorise where their favourite weapon is. If a bot has success with a particular weapon, they should be more inclined to search for that weapon. Using data from the model, the bot can therefore decide which weapon it wants to find the most.

The third decision that is influenced by the model is deciding to chase or flee an opponent. When in combat, bots may decide they no longer wish fight a particular opponent, and flee. When an opponent flees, the other player must decide to either chase the opponent, or move on. Information about the success rate of the bots current weapon can be used to influence this decision. If the bot has been very successful against the particular opponent, they will be more likely to chase the fleeing player. Conversely, if the bot is engaged in combat with a bot who is much more skillful than the bot itself (given from

data in the model), the bot will be more likely to flee. In this way, the bots decisions to chase or flee are influenced by the skill of the opponent. This is similar to how a human player would react in this situation.

The ultimate goal for adaptive computer games is to adapt to any playing style of a human player. In some non-adaptive games, cunning human players will often find a strategy that works especially well. There are often weaknesses in AI systems, and once a weakness is found the human player can continuously exploit it, thus reducing the challenge of the game. An adaptive AI system however, would realise some weakness is being exploited, and attempt to alter its play to negate the weakness. In another case, when the human player attempts to exploit the weakness, they may be exposing themselves to a particular strategy, that if the AI could work out, would force the human player to change their strategy. If a game could do this throughout the length of the game, and continue to force the human player to try different strategies, it would likely be much more fun to play, and to play repeatedly. It is also likely that humans do some sort of opponent modeling while playing a game, so computer-controlled characters that use opponent modeling may appear to play more like a human player, which was one of the original goals of the Quake 3 Arena bot.

Because the bots in Quake 3 are designed to play like human players, and because of the relatively simple design of our model, we treat the human player in the same manner as a bot. When we refer to bots modeling opponents, we mean both other computer-controlled opponents, and human

opponents. Human players also choose which weapon to use during combat, and the desired outcome is exactly the same. Thus if a bot can model another bot, they can also model a human player. This allows us to perform automated testing, without the need for human participants. This thesis investigates whether bots in Quake 3 can model other bots, and use the information to increase their effectiveness. Because human players make exactly the same decisions as we have chosen to model for the bots, the ability to model opponents should transfer seamlessly to human opponents.

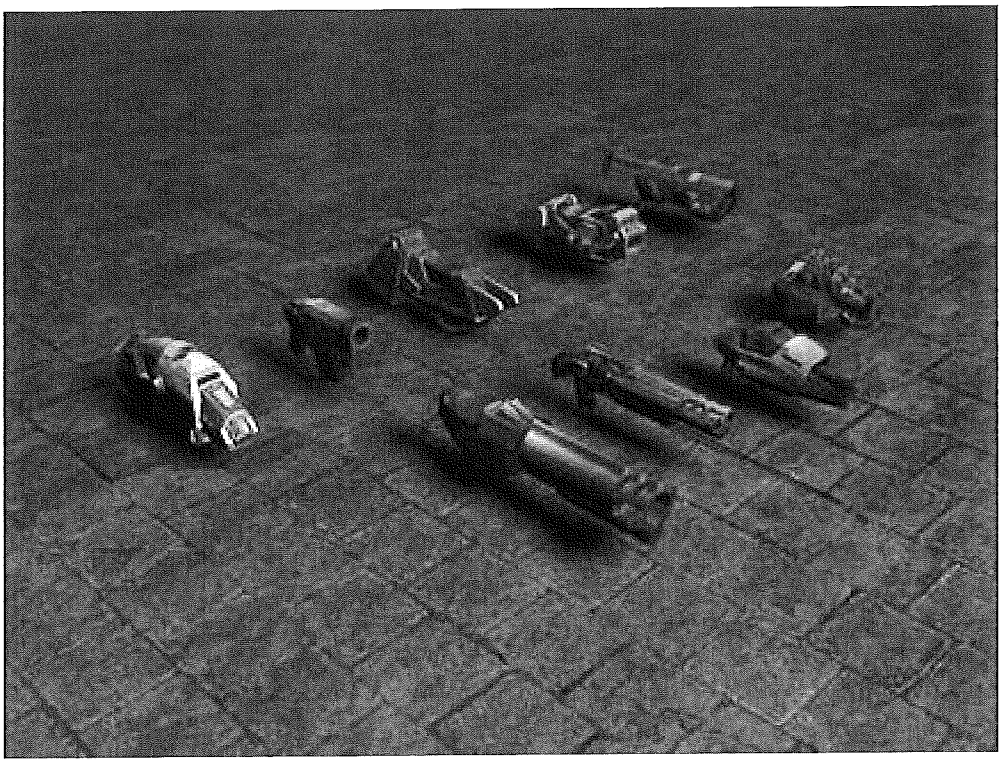


Figure 3.1: The weapons of Quake 3 Arena

Figure 3.1 shows the weapons available in Quake 3. They are: The

BFG10K, Grenade Launcher, Lightning gun, Plasma gun, Railgun, Rocket Launcher, Shotgun, Machinegun, and Gauntlet. The Gauntlet is a special hand-to-hand weapon, that requires no ammunition. Players generally only use this weapon if they have no ammunition for any other weapon. When a player spawns, they automatically have the Gauntlet and the Machinegun, with some ammunition. The other weapons must be picked up at given locations on the map. A player can carry as many weapons as they wish, which leads to decisions during combat.

Our opponent models consist of the history of these decisions between every pair of opponents. A bot will have an opponent model for each opponent it has ever engaged in combat with. Each time an interaction between the bot leads to a kill, the bot records which weapon they used, and whether they scored the kill, or the opponent killed them. All interactions between the two bots that lead to a kill are considered similar situations. These situations are divided by the weapon the bot was using at the time of the kill. The opponent model shows which weapon is most successful against the particular opponent. A weapon that has resulted in many desired outcomes, i.e. killing the opponent, is likely to be a better decision than a weapon than has lead to many undesirable outcomes. A reader familiar with First Person Shooter games may point out the same weapon, against the same opponent, will be better in some situations, and worse in others. This is a fair point but the complication lies in modeling these differences. Generally, the environment differentiates these situations: distance and obstacles be-

tween opponents may influence which weapon is most suitable. Attempting to model these environment variables complicates the model considerably, and also complicates using the model. One must define rules to identify similar environment situations, and also rules to govern decisions made in a given environment situation. This thesis focuses on a simpler approach, and aims to determine whether it can still be beneficial.

Once our model is defined we can decide how we wish to use the information. The obvious game decision to influence is choosing which weapon to use. For a given opponent, the opponent model shows which weapon has been most successful in the past; this weapon should be chosen, if possible. There are also two more decisions that can be influenced by the information in the model. These are choosing weapons to look for, and deciding when to flee or chase an opponent. Weapons not currently held are picked up at locations on the map. These locations do not change, so the bot may already know where a particular weapon is. How much they want a weapon they do not hold should be influenced by the opponent models. If the bot has a current opponent, they should look to pick up weapons that have been useful against that opponent in the past. Weapons may be picked up while chasing or fleeing an opponent. If the bot does not have a current opponent, their decision for which weapons to seek should not be influenced by any models.

The third use of the model is when deciding to chase or flee an opponent. When engaged in combat, a bot may decide to flee. The bot will do this if their health is too low, or they hold a weak weapon. For this purpose, the

weapons are ranked from weakest to strongest, and a bot is more likely to flee if they have a weak weapon. This algorithm exists in Quake 3, and is easily enhanced by opponent modeling. The ranking for the current weapon is combined with the information from the opponent model for that weapon. The opponent model may show that the current weapon is in fact successful against the opponent, in which case the bot will be less likely to flee. In Quake 3, the bots are continuously evaluating the best weapon to choose. If a bot flees because they are currently holding a weak weapon, they will quickly choose a better weapon, if possible, which will immediately influence their decision to continue fleeing. This prevents bots fleeing when they have a good weapon in their inventory, but are not holding it. When deciding to chase an opponent, the same logic is used. If the bot has a weapon that is relatively successful against the current opponent, they will be more likely to chase that opponent.

In summary, modeling which weapon was used and the outcome that occurred, for all combat situations that resulted in a kill or a death, gives us enough information to influence three tactical decisions made during gameplay. These decisions are which weapon to choose against a particular opponent, which weapon(s) to search for and whether to chase or flee an opponent.

3.4 Implementation

This research has added opponent modeling to Quake 3. Bots (computer controlled characters) model opponents in real time, as they compete in a match. The design of the model is shown in Figure 3.2.

```
models lucy
{
  opponent ranger
  {
    GAUNTLET 0 0 0 *0.73*
    MACHINEGUN 0 *0.9*
    SHOTGUN 1 *1.10*
    GRENADELAUNCHER 0 *0.9*
    ROCKETLAUNCHER 1 1 0 1 *1.2*
    LIGHTNING 0 0 1 *0.89*
    RAILGUN 1 *1.1*
    PLASMAGUN 1 0 0 0 *0.8*
    BFG10K 1 *1.1*
  }
  ...
}
```

Figure 3.2: An example of an opponent model in Quake 3

Figure 3.2 shows a section of the opponent models for the bot *Lucy*, showing the model for the opponent *Ranger*. Each bot will have a similar model for every opponent they have ever fought. The models are stored on disk, and grow over time. When a bot is loaded for a match, the model file for that bot is read into memory. Typically a match consists of up to eight players, and involves only a small subset of possible opponents. After reading in the models file, the relevant models are added to an in-memory

data structure, for fast access.

For each opponent, a history of interactions between the opponent and the bot is recorded. An interaction is defined as a kill or a death. During a match other interactions may occur, such as a hit that does not kill the target, or a shot that misses. It is however hard to define the meaning to the result of such interactions. For example, if a bot shoots and misses at an opponent, is it because the bot was using the wrong weapon, or was it just attempting a very hard shot? Similarly, if a bot hits an opponent, but does not kill them, is that a bad thing (they did not kill their opponent), or a good thing (they were good enough to hit them).

The history is stored per-weapon. The nine weapons listed in Figure 3.2 are the only nine weapons available in Quake 3 (not considering game expansion packs). For each weapon, a history of kills and deaths is recorded, modeled by ones (kills) and zeroes (deaths). A kill occurs when the bot hits an enemy, and reduces their health below zero. When the bots own health is reduced below zero, a death is recorded, against the opponent who last hit the bot.

When a bot scores a kill, it looks up its model for that opponent. If the bot does not have a model for that opponent, a new model is created. When the kill is recorded, a "1" is added to the corresponding weapon history of the model for that opponent. In the case of a kill, the 1 goes with the weapon the bot who scored the kill had, at the time of the kill. This is meant to reflect the bot's success with this weapon, against this opponent.

Conversely, when the bot itself is killed, the bot looks up the model for the opponent that killed it. Again, if no model is found, one is created. This time, a "0" is added to the model. The 0 corresponds to the weapon the bot had, at the time they were killed. In this design, no explicit information about the opponent is used, rather just what *this* bot had at the time of the kill or death. This design models how well the opponent dealt with the current tactics of the bot.

Applying this to the model in Figure 2, we can see *Lucy* has killed *Ranger* eight times, and been killed by *Ranger* eleven times. The model is small, so does not contain much information, but it looks like the *Rocket Launcher*, *Railgun*, and *BFG10K* have been reasonably successful against *Ranger*.

Following the history of interactions is the success value, shown in between two * characters. This is so the parser knows when the history stops and the success value starts. We describe how the success value is determined below.

As the bots play, the models grow, and over time eventually show how successful (or not) each weapon is against each opponent. A weapon with lots of 1s corresponding to it has been successful in the past, so is probably a good choice, as opposed to a weapon with lots of 0s, which has not been very successful.

During a match, every bot is constantly evaluating and deciding what they want to do. One aspect of this is what weapon to hold. Other aspects are concerned with goals such as *seek weapon*, *seek health* etc. Opponent models are used in two specific aspects of decision making: choosing a weapon,

including weapons to seek, and deciding whether to chase or flee an opponent. These decisions were chosen because they are related to opponents and weapons. The information in the model is limited to the success of different weapons, so is therefore only helpful when making decisions regarding opponents or weapons.

For each weapon, a success value is calculated. This success value represents the ratio of ones to zeroes for that particular weapon, against the particular opponent. Every time the model is updated this success value is recalculated, but only for the weapon that was updated. When the model is read, only the success values need to be read, saving time and memory. This proves efficient, as there are many more reads than updates.

The success value is calculated as follows: starting with a default value of one, parse the history for the weapon. For each one encountered, multiply the success value by 1.10. For each zero, multiply the value by 0.90. This simple algorithm effectively adds positive or negative 10% to the success value for each interaction. A weapon with a lot of ones in the history will have a high success value (higher than one), and a weapon with a lot of zeroes will have a low success value (less than one). A weapon with no history has a success value of one. 10% was chosen because it is big enough to make a significant difference after a number of interactions (10 kills with no deaths will have a large influence on choosing that weapon), but small enough not to give too much influence to lucky kills or deaths.

The bots in Quake 3 have a weight for each weapon, as discussed in

section 2.1.1. These weights are different for each bot, and represent the bots preference for that weapon. When selecting a weapon, the bot simply chooses the weapon with the highest weight, assuming the bot currently holds the weapon, and also some ammunition for that weapon. The bot will always make the same decision for every opponent.

With opponent modeling, the weight for each weapon is first multiplied by the success value of that weapon against the current opponent. These modified weights are then compared to select a weapon. The weapon with the highest modified weight is selected. This explains why the default success value is one, as the weapon weight will be unchanged. With our opponent modeling implementation, bots select weapons based on a combination of the success value for every weapon against the current opponent, and the bot's own preference for each weapon. This means the bots still act differently (the purpose of the weights), but combine that with knowledge about the opponent.

The opponent model is also used when the bot must decide if they want to chase a fleeing opponent, and if they themselves want to flee an opponent, or keep fighting. When deciding to chase or flee, a value representing the bot's "Aggression" is calculated. If the bot's aggression is greater than 50 (range 0-100), the bot will want to chase. If the aggression is less than 50, the bot will want to retreat (flee). This strategy exists in the original game, but is enhanced in our version of Quake 3 to use information from the opponent models. The existing aggression calculation is extended by multiplying it

with the success value for the current weapon against the current opponent. Multiplication is used because the success values are often close to a value of 1. A weapon must have a success value of 2 to double the bot's aggression, which only occurs if the bot has been very successful with that weapon.

3.5 Summary

In this section we gave the motivation, theory, design and implementation of opponent modeling in Quake 3. We aim to make the computer controlled characters (bots) in Quake 3 smarter, and provide them with a method to adapt to a human players tactics. With this enhancement, the bots record a history of every interaction between themselves and an opponent that results in a kill or death. A kill occurs when the bot reduces the opponents health below zero, thus killing them, and scoring a point. A death occurs when the opponent reduces the bots own health below zero. Recording these interactions, and which weapon was used at the time of the kill or death, provides the bot with data reflecting the skill of the opponent, and which weapons are most and least effective against that opponent.

Once collected, the data from the model is used to influence three tactical decisions. These decisions are which weapon to choose, which weapon(s) to search for, and whether to flee or chase an opponent. Which weapon to choose and which weapon to search for comes directly from the information in the model. This information is combined with the bots own static preferences to

give weights for each weapon. The bot chooses the weapon with the highest weight, both to use and to search for. The bot combines their preference for each weapon with information about how successful that weapon is against the current opponent.

Deciding to flee or chase an opponent similarly combines the existing algorithm with information from the model. In the original algorithm, the weapon the bot currently holds is assigned a value, related to how powerful the weapon is. If this value is above a threshold, the bot will chase an opponent that flees; if the value is below the threshold the bot will flee from combat. Both the value assigned to the weapon and the threshold is static, and identical for all bots. This means a bot will be just as likely to flee from a weak opponent than from a skilled opponent. With the opponent modeling addition, the existing value for the current weapon is combined with the success value for that weapon, for the current opponent. The result is that a bot will be more likely to chase a weak opponent, and flee from a skillful opponent. This behaviour both makes theoretical sense and provides practical benefit, as shown in the next section.

Chapter 4

Evaluation

4.1 Evaluation Criteria and Design

The purpose of this research is to make the bots in Quake 3 Arena smarter. This is achieved by effectively giving the bots *memory.* The bots remember what worked well in the past, and use this information to influence their strategy. The result is that a bot who uses opponent modeling should score more points, by killing more opponents, than if they were not using opponent modeling. The bots score can then be used to measure the benefit realised from opponent modeling.

Although opponent modeling is intended to be used against human players, the bots in Quake 3 have been specifically designed to play like a human player, as previously mentioned. Evaluation can thus be performed between computer characters, without the need for human participants. The infor-

mation modeled by the computer characters is also low-level enough that the same information can be extracted from modeling computer characters, as would be from a human player. Computer players also must choose a weapon from the selection of available weapons, and kills and deaths occur in the same manner. Computer players competing in matches against one another is therefore used to evaluate the effectiveness of opponent modeling in this project.

The benefit of this is that many more test can be performed, without requiring human participants. It also reduces some variability naturally introduced with human volunteers. Opponent modeling requires a period of time before the model is accurate, and this makes experiments with human participants troublesome. A player may need to play the game for an hour or more before the true benefit of the model can be seen. This problem is compounded by the fact that every match in Quake 3 is different. The same bot will get a different score playing the same opponents, on the same map. Due to this variability, to get an accurate measure of the bots skill, many repetitions must be performed. Testing with exclusively computer characters allows for more tests to be performed, and helps keep each test consistent.

In Quake 3 all the bots are slightly different, in both characteristics and weapon preferences. The result of this is some bots perform better than others. It would be inaccurate to compare a bot using opponent modeling with a bot who does not use modeling, as there are other differences. Instead, a bot without modeling is compared to the same bot, but using the modeling

technique, in a different match. This shows how much difference, in terms of kills, the opponent modeling implementation makes. Many iterations of the same match are performed to ascertain averages in both circumstances. The average score for the bot when not using modeling is compared with the average score for the same bot when using modeling.

4.2 Analysis of scores

Matches in Quake 3 can involve a variable number of players. Some small maps are designed for one-on-one action, while some larger maps can involve eight or more players. Various match conditions are evaluated in this section, including using different maps, and different numbers of players.

4.2.1 Matches between two players

The first test involved two bots: *Mynx* and *Orbb*. Ten matches were played with a score limit of 100; the first bot to score 100 kills is the winner. Neither bot was using opponent modeling in these first ten matches. Figure 4.1 shows the scores of the two bots in each of the ten matches. Mynx is the superior player, winning every game. Orbbs score fluctuates around 40.

Orbb was then given opponent modeling, but no existing model. Orbb would have to build the model while competing in the next ten matches. The scores from these ten matches, with Orbb using opponent modeling are shown in Figure 4.2. The same model was used and built upon in all ten

matches for the second set.

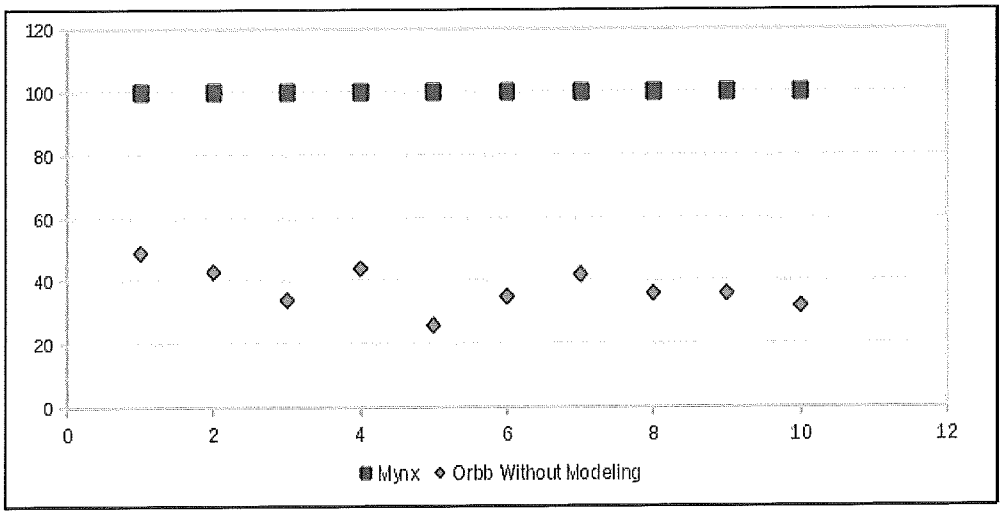


Figure 4.1: Scores when no opponent modeling is used

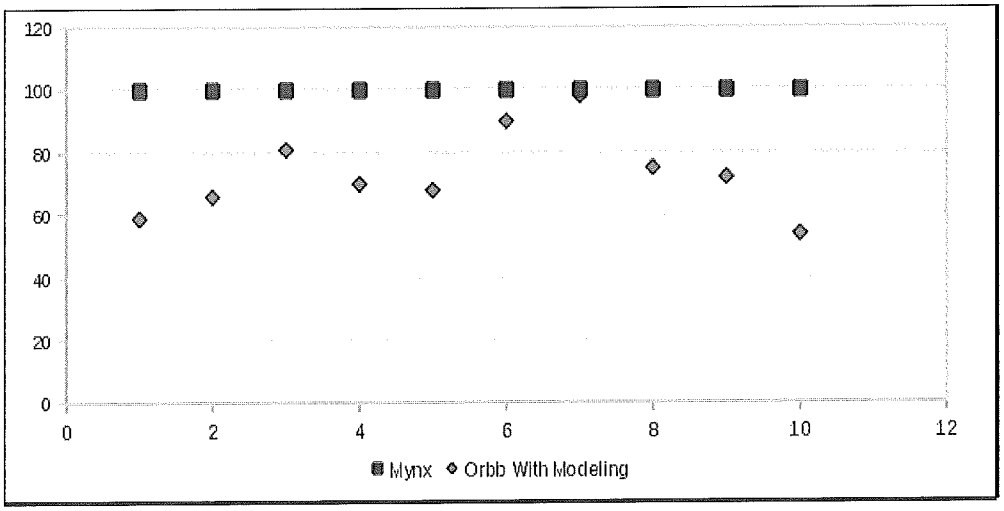


Figure 4.2: Scores when Orbb uses opponent modeling

The scores for Orbb in the second set of matches are significantly higher (t=-7.5, p=0.000002) than in the first set. The only difference between the two sets of matches in that opponent modeling was enabled for Orbb in the second set. These results show that this opponent modeling technique does improve the bots performance. It improves the bots performance by remembering what worked well in the past, and making decisions based on what the bot knows about its opponent. Although Mynx still wins every match, Orbb is much more competitive, and very nearly wins in match seven. It appears that after two matches the model is sufficiently accurate, although even in the first match Orbbs score is much better than in any of the matches from the first set. There is a general upwards trend, with the highest score being in match seven. Interestingly, the lowest score is in the very last match. This may show that the model has become less useful, or it may be a coincidence. Importantly, in every match in the second set, Orbb gets a better score than *any* match from the first set.

Figure 4.3 shows Orbbs scores for both sets of matches. The improvement realised by opponent modeling is immediately visible. Table 4.1 shows the average score and standard deviation from the two sets of matches. Mynx's score are not included as she scored 100 in every match. The average score when Orbb uses opponent modeling is twice that of when no modeling is used. The standard deviation is also much higher, showing that Orbb's scores are more variable when opponent modeling is used.

	Average (n=10)	Standard Deviation
Orbb (no modeling)	37.7	6.75
Orbb (modeling)	73.3	13.43

Table 4.1: Average score and Standard Deviation for Orbb

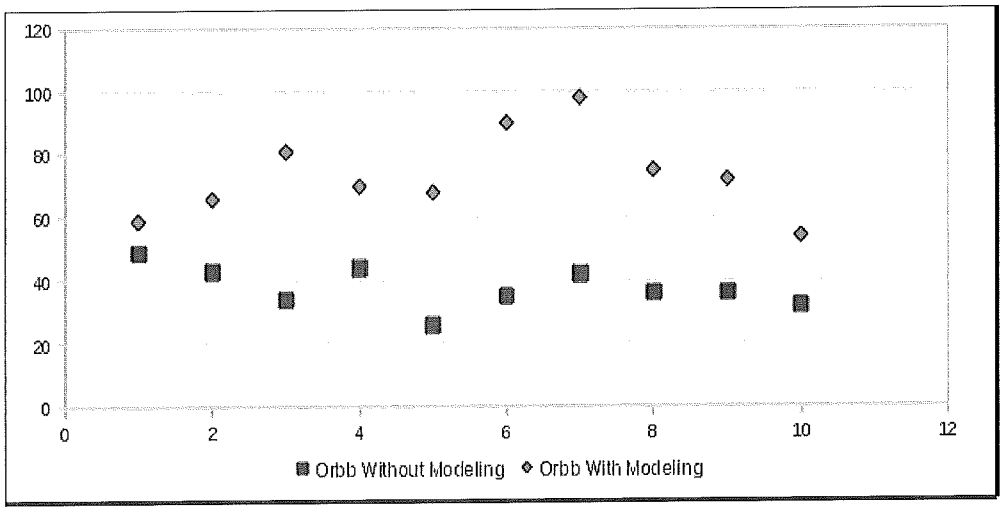


Figure 4.3: Orbb's scores, with and without opponent modeling

After these initial positive results, a further test involving Mynx and Orbb was performed. This time, the opponent modeling implementation was modified to use less information from the model. Namely, the aggressiveness of the bot was not influenced by the information from the model. In the original Quake 3 game, bots decide to chase or flee and opponent based on which weapon the bot itself has. If the bot has a more powerful weapon,

they will be more likely to chase a fleeing opponent, and less likely to flee from an opponent. With opponent modeling, this algorithm was enhanced to include information from the model. The success value against the current opponent, for the weapon the bot currently holds is used as a multiplier for the aggression calculation. With this enhancement, a bot will flee from skilled players more, and will be more likely to chase weak players. For this test, this addition to the aggression calculation was removed, leaving the original Quake 3 algorithm. The model construction and weapon selection algorithms were left as in the previous tests. This test ran over 20 matches, to see if the models effect changed at all over a longer period of time. Figure 4.4 shows the scores for the twenty matches.

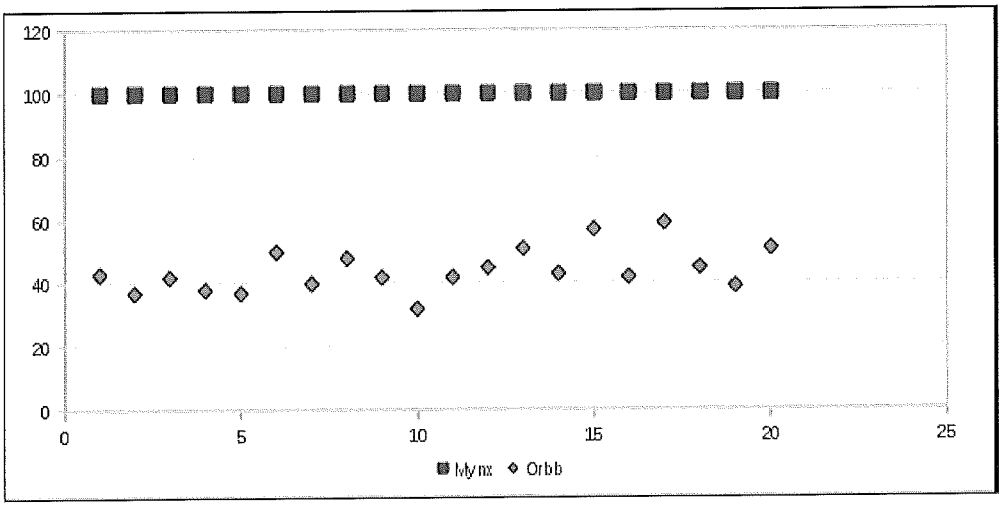


Figure 4.4: Bot Aggression not effected by models

When the bot aggression calculation is not influenced by information from the opponent model, the effect of opponent modeling is reduced considerably. Table 4.2 summarises the differences. As Mynx is the superior player in the tests, one could presume that Orbb is fleeing more from Mynx when full opponent modeling is used. It is important to note that to score points one must kill an opponent; staying alive is not so important. It seems that fleeing a stronger opponent more often actually results in more kills against that opponent. This is an interesting observation, although it makes some sense. Fleeing an opponent allows time to gather health and additional weapons (although your opponent gains time for this as well). This behaviour mirrors what a human player would do: when in combat, a human player will be much more likely to flee from an opponent they know is very skillful, and also more likely to chase an opponent they know is less skillful. In this regard, opponent modeling makes the bots act more intelligently, like a human player, resulting in a better score. In this test the average after twenty matches is higher than the average after the first ten matches. This may suggest that the models are slowly becoming more useful, or may reflect the variability of the game.

	Average
Orbb, no modeling, n=10	37.70
Orbb, modeling, n=10	73.30
Orbb, aggression not effected, n=10	40.90
Orbb, aggression not effected, n=20	44.15

Table 4.2: Average scores for Orbb

4.2.2 Matches between six players

In Quake 3 many matches involve more than two players. The next test was performed on a different map, with six bots. This was done to evaluate opponent modeling in a different context. A match against a single opponent is very different to a match against five opponents. With five opponents, much less time can be spent collecting weapons and ammunition. A player's score will also depend on the distribution of skill within the group. If one player dominates, the scores for all other players will be lower than if all players are evenly matched. Because the matches end when the leader scores a certain number of kills, the other bots get more or less time to score points depending on how long it takes the best player to win. In this test, all six bots initially played without any form of modeling, to gauge the relative skill of each character. Opponent modeling was then enabled for two bots, and the results compared. Only two bots were given opponent modeling because it was assumed that if all bots were given opponent modeling, they would all perform better by relatively the same amount, resulting in the same scores. For this test each set of matches consisted of 20 matches. From the previous tests it appeared that ten matches might not give an accurate average of a particular bots skill. It was also assumed that matches of six players would result in more variable scores than matches between two players, hence the higher number of required iterations. Again, each match ended when one

player reached 100 kills. Table 4.3 shows the average score and standard deviation for each of the bots. Figures 4.5 and 4.6 show the scores for the bots who used opponent modeling in the second set of matches (Gorre and Wrack).

	No Modeling		Modeling (Gorre & Wrack)	
	Average, n=20	Std. Dev.	Average, n=20	Std. Dev.
Slash	98.65	4.38	98.40	5.23
Gorre	80.05	11.25	81.35	11.29
Wrack	70.20	13.32	77.75	13.89
Patriot	66.40	18.70	74.05	14.59
Biker	65.90	9.70	65.80	9.47
Lucy	66.90	10.40	67.85	10.29

Table 4.3: Average scores for all players

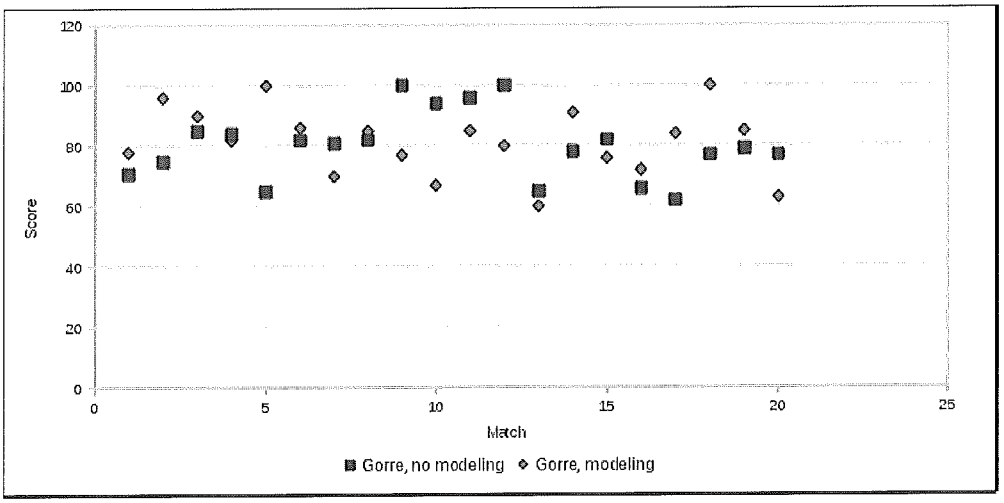


Figure 4.5: Gorre's scores

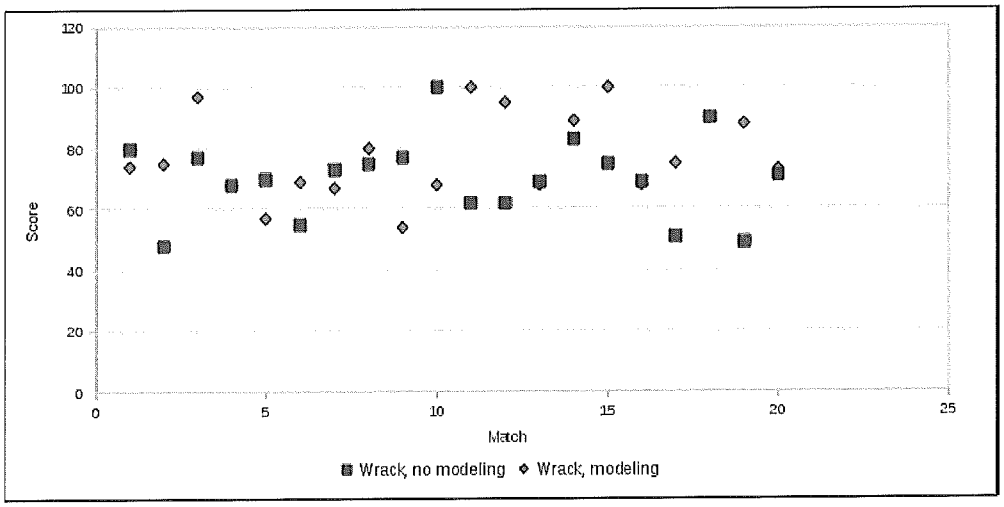


Figure 4.6: Wrack's scores

When six bots compete in match the effect of opponent modeling is less clear. In general, Gorre's scores are higher when using opponent modeling, but not significantly. Matches 9, 10, 11 and 12 from the first set of matches are in contrast to the rest of the results, and reduce the affect. When not using modeling, each match is independent of anything before, so the fact that there are four high scores in a row in coincidental. There is no upwards trend when using opponent modeling, so it seems the model is as useful as it gets at the end of the first or second match. The range of scores when using opponent modeling is quite similar to when no modeling is used.

Wrack's result's, shown in Figure 4.6, are significantly higher when using opponent modeling (t=-1.75, p=0.04). There are more scores at the high end of the scale when opponent modeling is used; six scores higher than 80 compared to only three scores above 80 when opponent modeling is not used. When opponent modeling is used, there are only two scores below 60; with no modeling there are four.

There are a few possible explanations for these less visible results. When six bots compete in a match the score for any one bot depends in a way on the scores of the other bots. If a bot is getting killed a lot, it is more difficult to score kills, with or without a model. Also, as there are six bots on the map, there is less opportunity to gather weapons and implement tactics. The games with six computer players took only half as long as the games with two players, even though the six players fought on a bigger map. Also, a bot facing five opponents has to do five times as much work (scoring kills) to build adequate models than a bot only facing one opponent (who only needs to build one model).

As shown in Table 4.3, both Gorre and Wrack (the bots who had opponent modeling enabled) had a better average score in the second set of matches, with Wrack's increase being significant. Patriot's score also increased although no modeling was used, but this increase was not significant. The averages for the three other bots stay almost identical. Patriot's score may be due to the relationship between the bots during the course of a match. It is possible that as Gorre and Wrack were doing better, Slash (the usual

winner), was doing relatively worse. As the game ends when a player reaches 100, all the bots have an opportunity to score more kills if it takes the best player longer get there. The games that took the longest tended to result in the highest overall scores of all the bots, and were the most even. The relationship between the bots scores makes it hard to isolate the effect of opponent modeling, which is already likely reduced by the increase in game speed due to six bots competing at once, in what can certainly be called a death-match. However, the only bot to show a significant change in performance was one of the bots given opponent modeling, showing the technique influenced the bots performance in a positive manner.

4.2.3 Matches between four players

The next test involved four bots, to see if the effects of oppon*Mynx*ent modeling would be more visible. The matches were also played on a different map, to see if opponent modeling was still useful. In the same manner as the previous test, the bots competed in 20 matches, to determine the relative skill on each bot. Two out of the four bots were then given opponent modeling, and other 20 matches were played. Table 4.4 shows the average scores and standard deviations for each bot.

	No Modeling		Modeling (Grunt & Wrack)	
	Average, n=20	Std. Dev.	Average, n=20	Std. Dev.
Daemia	97.80	5.57	98.45	5.24
Slash	82.05	13.48	75.15	17.54
Grunt	80.55	11.95	76.80	9.99
Wrack	67.65	12.64	68.70	16.33

Table 4.4: Average scores for all players

These results do not show a particularly positive effect from opponent modeling. Wrack's average score increases by only one, and Grunt's score decreases by almost four points. This would suggest that opponent modeling is only effective for some characters, and not others. Figure 4.7 and 4.8 show the scores for Grunt and Wrack respectively, both with and without opponent modeling.

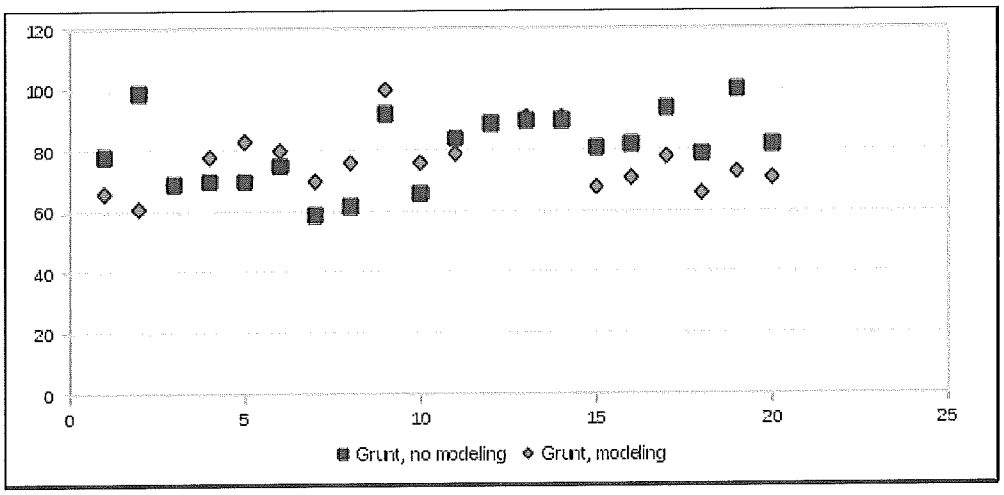


Figure 4.7: Grunt's scores

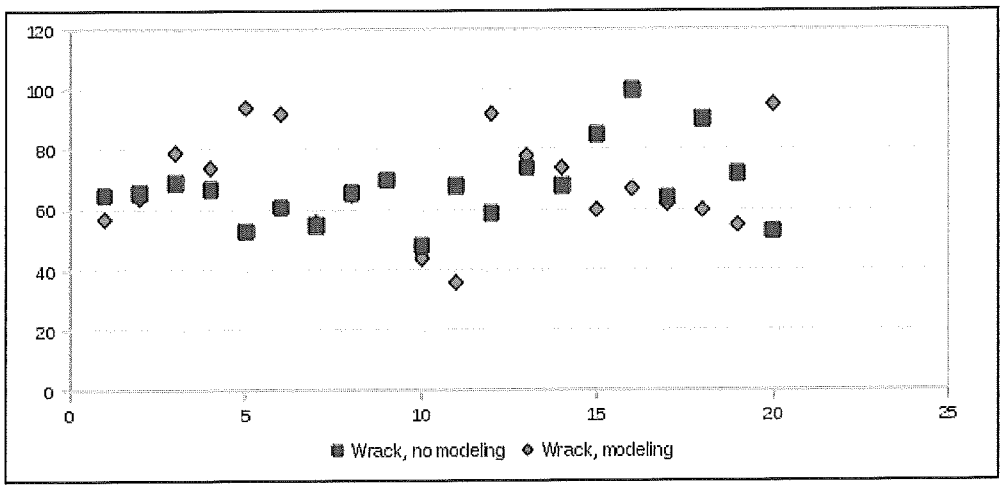


Figure 4.8: Wrack's scores

As Table 4.4 suggests, there appears to be little benefit realised from opponent modeling in these scores. However, further investigation into the data yields some interesting questions. Slash's average score is seven points lower in the second set of matches, when it was expected to remain consistent. This shows that factors other than opponent modeling, notably chance, can affect the scores in a considerable manner. Also, when not using modeling, both Grunt and Wrack seem to do better in the second half of the matches, compared to the first. There is no explanation for this, other than the variability of the game, as when no modeling is used every match is independent of any other match. Table 4.5 highlights this difference by showing the average score for all bots, when counting only the first 10 matches, and when counting all 20 matches.

	No modeling		Modeling (Grunt & Wrack)	
	Avg. n=10	Avg. n=20	Avg. n=10	Avg. n=20
Daemia	97.10	97.80	99.20	98.45
Slash	81.10	82.05	74.70	75.15
Grunt	74.00	80.55	75.90	76.80
Wrack	62.00	67.65	69.50	68.70

Table 4.5: Average scores with changing n

If only the first 10 matches are observed, the average score for Grunt and Wrack is considerably lower than if all 20 matches are observed. As noted before, this is interesting as all matches are independent when no modeling is used. For all the other six scores (Daemia and Slash in both sets, and Grunt and Wrack when using modeling), the averages change very little after 20 matches, compared to after 10. When not modeling, Grunt and Wracks averages change 6.55 and 5.65 points respectively, while no other bots average changes by more than one point, if the number of observed matches is modified. This discrepancy makes measuring the effect of opponent modeling difficult. The increase in performance for both Wrack and Grunt in the second 10 matches in the first set seems to be a coincidence with greater effect than opponent modeling itself. Figures 4.9 and 4.10 show the scores for Grunt and Wrack in the first 10 matches. Counting only the first 10 matches, Wracks increase is not statistically significant ($t=-1.35$, $p=0.10$).

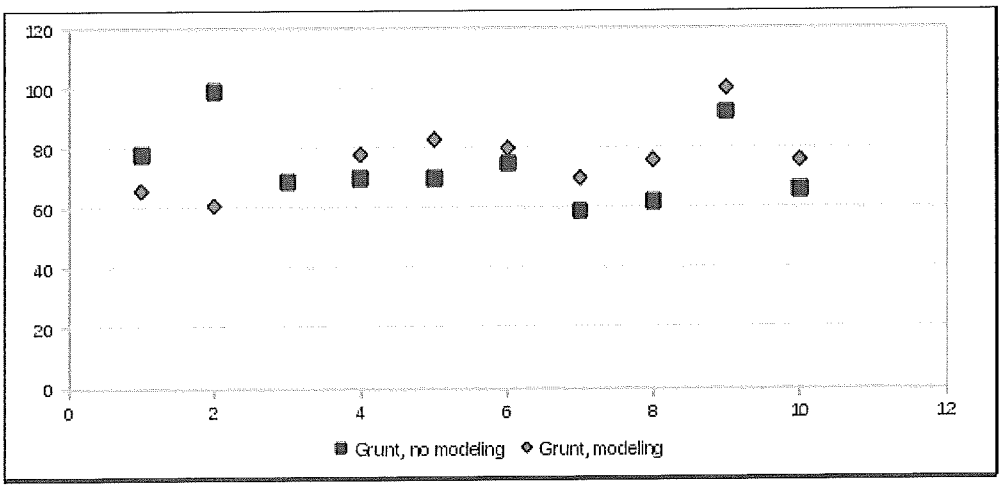


Figure 4.9: Grunts scores for the first 10 matches in each set

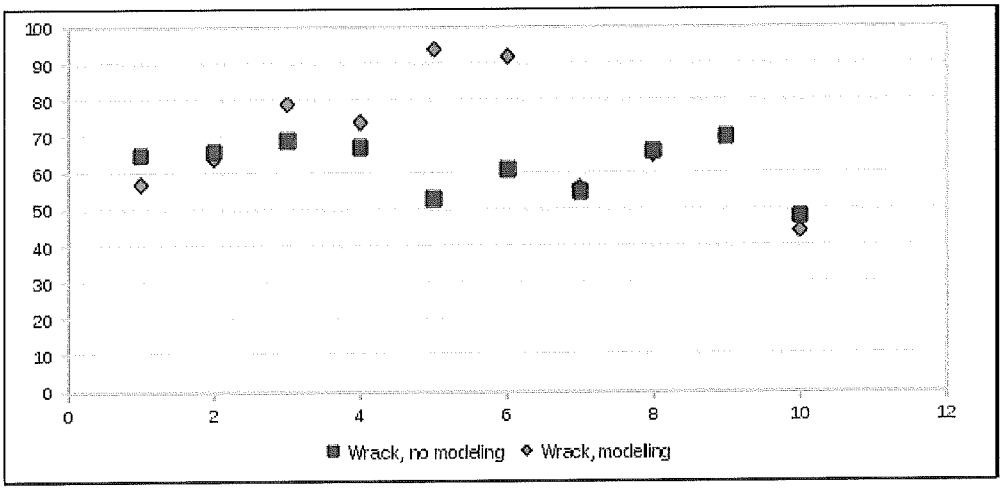


Figure 4.10: Wracks scores for the first 10 matches in each set

It is of course invalid scientific process to only count half the results from a test; Figures 4.9 and 4.10 are only given to show what would have occurred if only 10 matches were played in each set. Perhaps the first 10 matches in

the no modeling set were actually an under performance, and the second 10 were a more accurate reflection of the bots skill. If this is the case, opponent modeling has had little effect. If this is not the case, and the second 10 matches in the first set were coincidentally high, then opponent modeling would seem to have again had a positive effect, as shown by Figure 4.9 and 4.10. No conclusions can be drawn from this test however, except that it is inconclusive.

4.2.4 Evaluation of success value calculation

The next test attempted to measure the effect of opponent modeling on a finer level. First, two bots played each other 10 times, with no modeling. Modeling was then given to the weaker bot, and another 10 matches played. This time however, the model was deleted after each match (in previous tests the model was built upon in successive matches). After these 10 matches, a further 10 matches were played, this time building on the same model throughout the set of matches. A further 10 matches were then played, using a different success value calculation, to identify any different effects. For this test, bots *Daemia* and *Orbb* were chosen. Orbb showed success when given modeling against Mynx, so was chosen again to see if that success could be replicated against a different opponent.

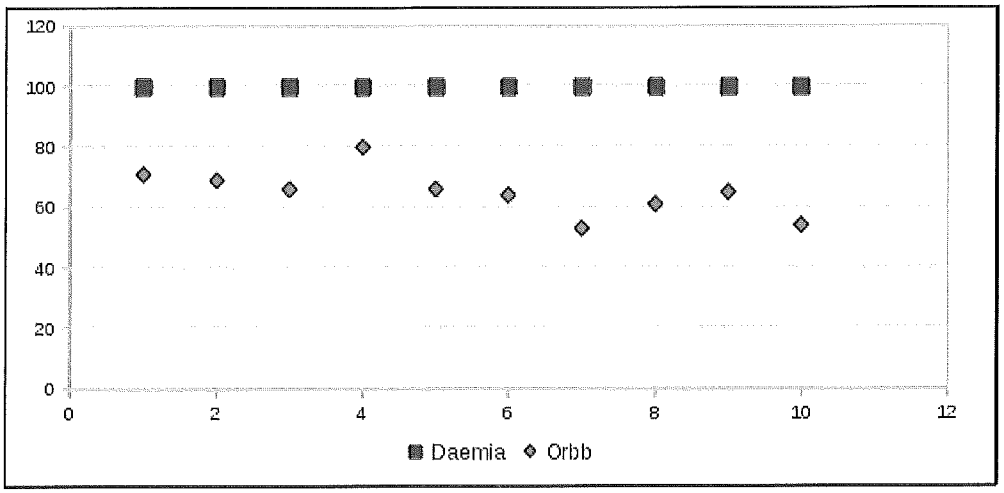


Figure 4.11: Daemia and Orbb, no modeling

Figure 4.11 shows the scores for Daemia and Orbb for each match, when no modeling is used. Daemia is the superior player, winning every match. Orbb was however much more competitive against Daemia than against Mynx, without opponent modeling. There appears to be a very slight downwards trend, although once again this cannot be explained as each match is independent. This may however suggest that the average score for Orbb when not using modeling, as calculated from these 10 matches is perhaps higher than it realistically should be.

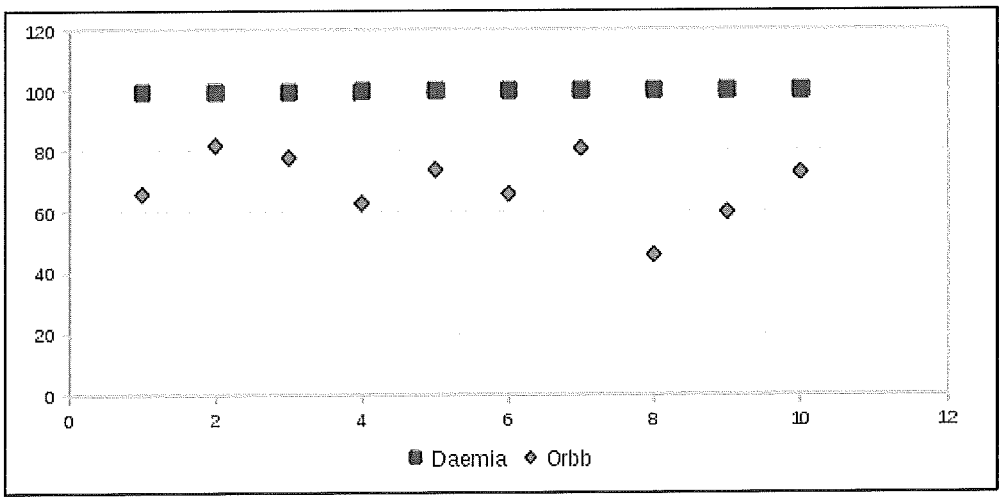


Figure 4.12: Daemia and Orbb, Orbb modeling, deleting models

Figure 4.12 shows the scores for both bots when Orbb is given opponent modeling. In this set of matches the opponent models were deleted after each match. Orbb had to build a new model for it's opponent in each match. It is worth noting that usually matches in Quake 3 have lower score limits than 100. A usual match in single player ends when someone scores 20 kills. Games with kill limits of 30 or 40 are reasonable, but regular matches of first to 100 are rare. Each match is therefor roughly equivalent to two or three regular matches. When given opponent modeling, Orbbs scores are on average higher than without opponent modeling. There is no trend, as expected, as each match in this set is independent of the others (because the models are deleted). This increase in scores however is not statistically significant (t=-0.67, p=0.25).

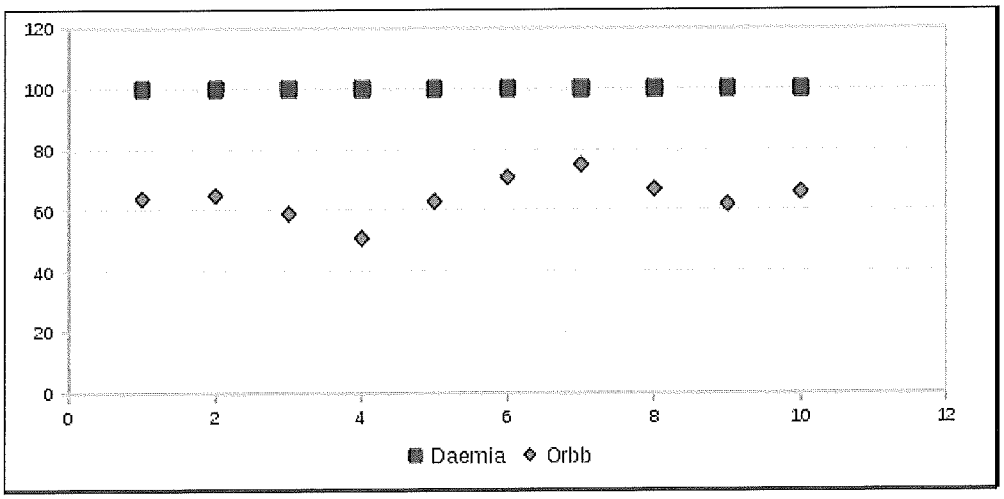


Figure 4.13: Daemia and Orbb, Orbb modeling, keeping models

Figure 4.13 shows the scores when Orbb is given opponent modeling, and is allowed to keep the models after each match. On average the scores are slightly lower than when the models are deleted after each match. However, there does seem to be a slight upwards trend, and the scores are more consistent. This is expected, as building a new model in each match results in models that are more affected by chance, rather than skill. This is because if a bot by chance does well in a match, the model will be quite different than if they performed poorly, even against the same opponent.

Figures 4.12 and 4.13 seem to suggest that the models are not as effective as they could be. The average score for Orbb is higher when the opponent models are deleted after each match, but one would expect for the model to become more useful, as it becomes more accurate. The opponent models usefulness depends on the accuracy of the success value, so a success value

calculation that is a more accurate representation of the bots skill should lead to the models being more useful. In all these tests, the same calculation for the success value of each weapon is used: Starting with one, for each *1* in the weapon history, add 10%. For each *0*, subtract 10%. For this test, a different calculation was evaluated, given below.

$$success\,value = ((ones + zeroes)/zeroes) - 1$$

where *ones* and *zeroes* are the total number of *1*s and *0*s in the history for a given weapon respectively. This simple calculation gives the ratio of zeroes to the total number of occurrences (ones & zeroes) in the weapon history. For a weapon with many zeroes (deaths), the denominator will be high, giving a low result. When there are few zeroes in the history, the numerator will be relatively higher, giving a high result. The -1 is to do with the fact that the success values are multiplied with values from the existing Quake 3 algorithms. A weapon with an equal number of zeroes and ones should return a success value of one. If *ones* and *zeroes* are equal, $(ones + zeroes)/zeroes$ will give two, hence the required subtraction of one.

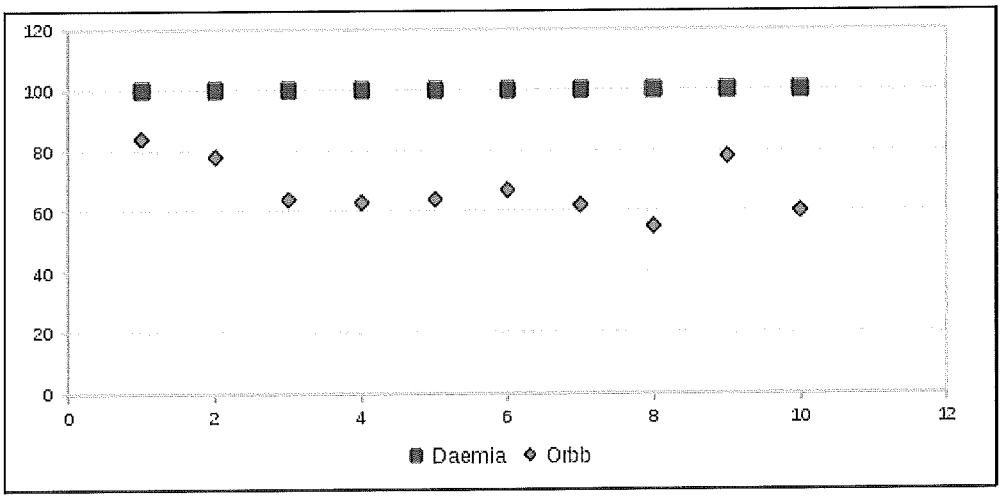


Figure 4.14: Daemia and Orbb, Orbb modeling, new success value calculation

When using this success value, Orbbs scores, shown in Figure 4.14, are similar to when the models are deleted after each match, and higher than when the models are retained, with the existing success value calculation. In this test, the models were also retained after each match, so it would seem this success value calculation is at least more successful for Orbb, when playing against Daemia. There appears to be a slight downwards trend, although if the first two matches are considered as outliers there is not an obvious trend, upwards or downwards.

Table 4.6 summarises and compares the average scores for Orbb for these four tests. Daemia scored 100 points in every match, so the table shows Orbb's scores only.

	Average (n=10)	Std. Deviation
No Modeling	64.90	7.89
Models retained	64.30	6.52
Models deleted	68.90	11.05
Modified calculation	67.50	9.31

Table 4.6: Orbbs scores for the four different tests

Table 4.6 shows that opponent modeling for Orbb against Daemia gives much less effect than against Mynx. This likely has something to do with the difference in scores when no opponent modeling is used. It appears that opponent modeling is more beneficial when there is a dramatic difference in the skill of the competing bots. The fact that the average score is higher when the models are deleted after each match may show the original success value calculation is less than optimal. It would appear that the models are indeed storing useful information, but improvement can be made on the way the information is extracted and used. This is reinforced by the modified success value calculation resulting in a higher score. It is important however to remember some observations from the previous tests, specifically the third test involving four bots. It may be the results have been skewed by abnormal performances. However, in all these tests opponent modeling does not make the bot worse (except for one bot in the troublesome third test). Opponent modeling therefore looks useful, in some cases more so than others. These tests show that modeling simple information such as a history of kills and deaths can influence the bots performance, although how best to use this information may still be unknown.

4.3 Analysis of generated models

In addition to reviewing the final score a bot achieves, it is possible to examine the resulting model. One would expect similar models to be generated for the same opponent, and dis-similar models for different opponents. To test the first hypothesis, namely that a similar model would be constructed for the same opponent, the models from the matches between *Daemia* and *Orbb* were used, from the test where the models were deleted after each match. It is expected that a similar model would have been constructed by the end of each match. The models were compared by inspecting the success values for each weapon, showing which weapon is most effective against a particular opponent. Which weapon is most successful should remain fairly consistent, although as with the other experiments the variability of the game has a considerable effect.

Table 4.7 summarises the generated models for Orbb, against opponent Daemia. For each match, the most successful weapon is shown in bold. In the cases where a weapon has a success value of 1.00 that weapon has not been used, so is not considered for the most successful. Unfortunately, the model for match eight was lost, so no data is available. The most likely reason for this is that the game crashed when writing the models out to disk. Interesting, in this match Orbb scored a low 46 (the average was 68.9). The low score and missing model may be related, unfortunately we will never know.

Match	1	2	3	4	5	6	7	8	9	10
Gauntlet	1.00	0.73	0.81	0.90	1.00	1.00	1.00	-	1.00	1.00
Machinegun	0.71	0.35	0.42	0.43	0.64	0.59	1.08	-	**0.57**	0.53
Shotgun	0.43	1.12	0.88	0.29	0.75	0.57	0.75	-	0.36	0.41
Grenade Launcher	0.56	0.89	0.70	0.59	**1.04**	0.51	0.66	-	0.36	0.58
Rocket Launcher	**1.34**	**2.65**	**1.32**	**2.65**	0.98	**0.80**	**1.19**	-	0.46	**3.27**
Plasmagun	0.13	0.18	0.47	0.16	0.16	0.15	0.27	-	0.39	0.16
Score	66	82	78	63	74	66	81	46	60	73

Table 4.7: Summary of models for Orbb

Table 4.7 shows that in seven out of nine matches, the Rocket Launcher is the most effective weapon for Orbb, against Daemia. The values for the other weapons are fairly consistent, especially for the Plasmagun. This result is confirmed in Figure 4.15, a graphical representation of the above data. The Lightning gun, Railgun and BFG10K have not been included as were not available on the map used for these matches.

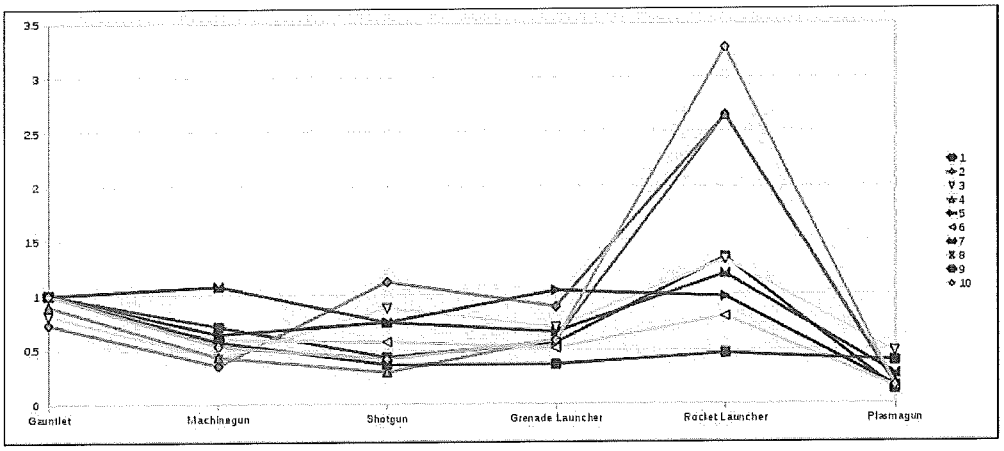


Figure 4.15: Success values by weapon for Orbb

These results suggest that a similar model will be constructed for the same

opponent. The models used for the data above were relatively short-lived models, as they were deleted after each match. Some variability is therefore expected. However, as only one opponent was being modeled, perhaps the opponent model was more a reflection of the bots skill with different weapons, rather than a reflection of a particular opponents weaknesses. To assess this possible situation, models for different opponents need to be compared. If models for different opponents are also similar, it would suggest this is the case, and effectively nullify the argument for opponent modeling. However, if the models are different, it would show that the models as least in part reflect something unique about the opponent.

Table 4.8 summarises the models for the bot Wrack, against five different opponents. These models are the final models after all tests were conducted, so include much more data. Again, the most successful weapon is indicated in bold.

	Biker	Gorre	Lucy	Patriot	Slash
Gauntlet	0.81	0.90	0.90	1.00	1.10
Machinegun	0.39	0.48	0.43	0.80	0.48
Shotgun	1.17	0.87	0.86	1.07	0.66
Grenade Launcher	0.99	1.10	0.89	**1.21**	0.99
Rocket Launcher	0.78	0.68	1.03	0.94	**1.25**
Lightning	1.10	0.89	1.10	0.81	1.00
Railgun	1.20	0.73	0.80	0.99	0.97
Plasmagun	**2.38**	**1.17**	**1.19**	0.34	0.73
BFG10K	1.10	0.99	1.09	1.09	1.10

Table 4.8: Summary of Wracks models against different opponents

Although these values initially seem similar, this is largely because the

range of values is small. The plasma gun is the most effective weapon against three opponents, although only by a small margin for two of these. Also, the plasma gun is in fact the least successful weapon against the opponent Patriot. An issue with comparing success values such as these is that some weapons are used much more than others. This means that for some weapons the success value is less accurate than for others. Also, as bots decisions are still affected by their static preferences, some similarity is expected. From Table 4.8, at least two suggestions for tactics arise: When possible, try to use the Plasmagun against Biker, and don't use the Plasmagun against Patriot. It is these such cases where opponent modeling gives benefit. When the success values are similar, the bots static preferences have relatively more influence. The opponent model is useful when it shows something that is either much more or much less effective than the status quo. Some weapons, such as the BFG10K and the Lightning gun are not found in many locations, so are available to use less often than other weapons. Figure 4.16 shows the same information, in graphical form.

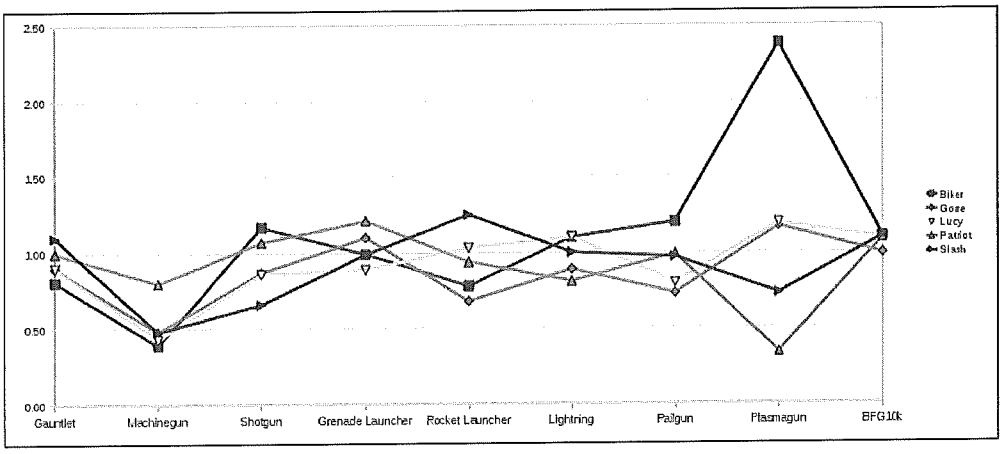


Figure 4.16: Success values by weapon for Wrack, against different opponents

Table 4.9 shows the same information, for the bot Gorre. The first observation is that the range of values is much greater: from 0.10 to 10. These are the minimum and maximum values for the success value of a given weapon.

	Biker	Lucy	Patriot	Slash	Wrack
Gauntlet	0.25	0.33	0.35	0.28	0.25
Machinegun	0.10	0.10	0.21	0.10	0.10
Shotgun	2.57	**10.00**	**10.00**	0.10	2.89
Grenade Launcher	4.98	1.71	2.81	0.23	1.05
Rocket Launcher	**10.00**	0.11	5.69	0.10	**10.00**
Lightning	0.90	1.10	0.89	1.00	1.10
Railgun	4.62	0.29	7.25	0.10	2.84
Plasmagun	0.24	2.30	0.82	0.10	0.36
BFG10K	9.37	**10.00**	1.63	**6.79**	2.39

Table 4.9: Summary of Gorres models against different opponents

Figure 4.17 shows the same information, in graphical form. The variability is immediately obvious: there seems to be little similarity between models of different opponents. The Lightning gun was again less available, so the

success values are close to the default value of 1.00. These results indicate that the models being constructed are indeed different for different opponents, therefor must be measuring something unique about the particular opponent.

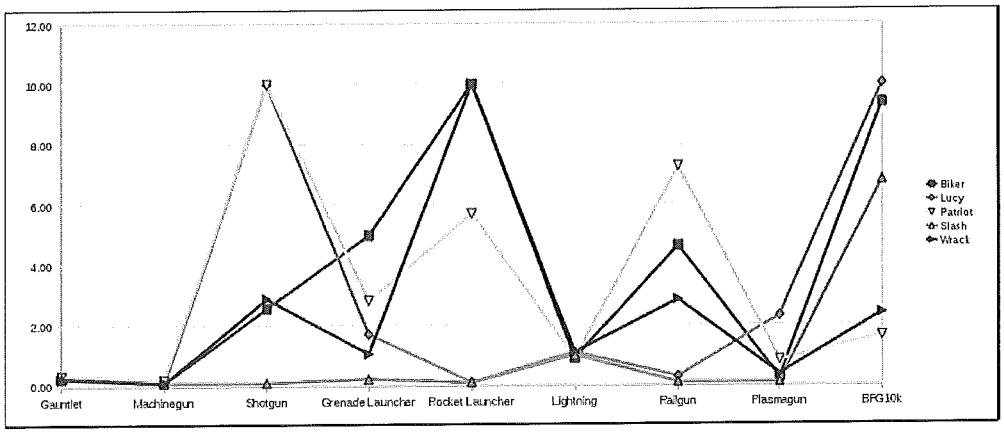


Figure 4.17: Success values by weapon for Gorre, against different opponents

One observation from these results, especially Table 4.9, is that a model for an opponent weaker than the bot itself is very different to a model for an opponent stronger than the bot. This information is useful, as shown in the experiment where the bots aggression was not influenced by the model. However, the effectiveness of using the model to select a weapon is reduced, especially when the success values for different weapons reach the minimum. It could be beneficial to therefore have two success values: an absolute and a relative value. The absolute value (the current success value) would be used for aggression calculations, and the relative success value used for weapon selection. The relative success value would be normalised against the other

weapons to negate the skill of the opponent. This would be a very interesting modification, but was not possible in this project due to time constraints.

4.4 Summary

The results from Section 4.2 and 4.3 show that opponent modeling can increase a bots performance, and that the models that are constructed are consistent for a given opponent. Although the model for a given opponent will be similar each time it is constructed, this is only true for the bot who is modeling the opponent. A different bot will construct a different model for the same opponent. This shows that the model is a combination of information about the opponent, and information about the bot itself. This makes sense, as the bot is using only its own experiences to construct the model. It makes sense that two different bots may each find a different weapon that works well against the same opponent. Bots have their own skills and weaknesses, and must find a weapon that works well for them, against a particular opponent. This is much like what a human player would do. The player must also learn what works well for them against a particular opponent; a weakness that cannot be exploited is not of much benefit. An aspect this evaluation did not touch on is how opponent modeling effects the playing *style* of a bot. Did they play more or less like a human player? The problem is that this is very hard to measure. One must first define how a human player plays, then attempt to gather data about the bot in a way

that facilitates comparison. Although difficult, this is entirely possible, and could give interesting results. However, this was not conducted as part of this project due to time constraints. It was chosen not to perform a study with human participants in this project also largely due to time constraints. It was felt that many iterations, and therefore much time, would be required to get even roughly accurate results, and it is assumed that the results would not be significantly different to testing with exclusively computer characters. However, given the results shown in the above evaluation, it could be very interesting to perform these further studies at a later date.

Chapter 5

Conclusion

5.1 Opponent modeling in Quake 3

This research was inspired by student modeling techniques from Intelligent Tutoring Systems, and adapted to a first person shooter game. The aim was to make the computer-controlled characters more competitive, by remembering information about specific opponents, and selecting strategies based on this information. Such strategies are which weapon to choose, which weapon to look for on the map, and whether to flee or chase an enemy. These decisions are now partly based on the computer player's knowledge about the specific opponent. Evaluation results show that this simple user modeling implementation can have significant results, although the particulars of the game make conclusive evaluation difficult. The fast pace and variability of the game make it hard to isolate the changes caused by opponent modeling,

although positive effects can still be seen.

The opponent models are constructed by recording, for every combat that resulting in a kill or a death, the weapon the bot was using, and whether the bot scored a kill, or was itself killed. Over time this information averages out over all the variables in the game to give a value representing the bots proficiency with each weapon, against each opponent. The models have deliberately been designed to be general enough to be used in every combat situation, and are applicable to most First Person Shooter games.

5.2 Evaluation

Quantitative evaluation of this project was performed with many matches being played by computer controlled characters. This evaluation showed opponent modeling to increase a bot's performance significantly, although not in all cases. Tests were carried about between two, four, and six computer players. When two computer controlled characters played against each other, opponent modeling doubled the bots average score. When four bots played however, little effect was seen. Matches with six bots showed better results than with four players, but considerably less effect than with two players. It seems that opponent modeling is more effective for and against some bots, and less effective for and against others. The most significant improvement was seen in a one-on-one match, where one bot was much more skillful than the other. The weaker bot improved dramatically when opponent modeling

was enabled for it, but not for its opponent. Interesting, this result is often seen in Intelligent Tutoring Systems, where the weaker students tend to show the most gain. This may be due to the similarities in implementation, or simply because those cases have the most to gain from any improvement in the system. This is quite an acceptable result in the context of a First Person Shooter game, as it helps to eliminate the games where the opponent is not even skillful enough to be challenging. If we can make those opponents smarter, so they are at least more challenging, even if they do not win, opponent modeling has been successful.

5.3 Future work

The results suggest that modeling kills and deaths is useful enough to have a positive effect. A problem with opponent modeling is there is often so much possible information that can be modeled. The danger is that as the information modeled becomes more specific, the times were the information is relevant decreases. The implementation in this project effectively averages out over all the possible situations, and provides information that although in any one situation may not be terribly useful, over time has enough benefit to directly increase the bots performance. Although the information in the model is useful, the evaluation suggests that more work could be done on how to best extract the information. In particular, the success value calculation seems less than optimal, and would be an interesting area for further refine-

ment. The idea of an absolute and a relative success value, to be used by different aspects of the Artificial Intelligence component of the game engine is also an interesting path for research.

The evaluation of this project largely focused on quantitative results. There are however, qualitative benefits for opponent modeling, although these are much harder to evaluate. Decisions such as deciding to flee an opponent based on the skill of the opponent seem like natural strategic decisions, but were not part of the game engine until this project.

This is a simple concept and implementation, and in its current form could be incorporated into any first person shooter game. The design could be extended if desired for a particular game, which would most likely give larger effects. As graphics advancements and special effects become standard, game developers will have to find other ways to differentiate their games, and ensure they remain enjoyable for a long period. Adaptive AI could be an increasingly important aspect of a successful video game. Other games have implemented opponent modeling in different ways, often with mixed results. This research shows that simple but effective opponent modeling is possible in a First Person Shooter game.

Bibliography

[1] http://www.theesa.com/facts/salesandgenre.asp

[2] Origins and History of Unix, 1969-1995 in The Art of Unix Programming

[3] Ohlsson, S. Constraint-based student modeling. Journal of Artificial Intelligence and Education, 3(4), p. 429-447, 1992.

[4] Darse Billings, Denis Papp, Jonathan Schaeffer, Duane Szafron, Opponent modeling in poker. Proceedings of the fifteenth national/tenth conference on Artificial Intelligence/Innovative applications of artificial intelligence, p. 493-499, 1998.

[5] Richards, M., Amir, E., Opponent modeling in Scrabble. Proceedings of the Twentieth International Joint Conference on Artificial Intelligence, p. 1482-1487, 2007.

[6] F. Schadd, S. Bakkes, and P. Spronck, Opponent modeling in real-time strategy games. In 8th International Conference on Intelligent Games and Simulation (GAME-ON 2007), M. Roccetti, Ed., p. 61-68, 2007.

[7] Julian Togelius, Renzo De Nardi and Simon M. Lucas: Making racing fun through player modeling and track evolution. Proceedings of the SAB Workshop on Adaptive Approaches to Optimizing Player Satisfaction, 2006.

[8] Mitrovic, A., Martin, B. Evaluating the Effect of Open Student Models on Self-Assessment. International Journal of Artificial Intelligence in Education, Special issue on Open Learner Modeling. 17(2), p.121-144, 2007.

[9] Littman, M. Algorithms for sequential decision making. Technical Report CS-96-09, 1996.

[10] Yannakakis, G. N., Maragoudakis, M., Player modeling impact on players entertainment in computer games, in User Modeling, p. 74-78, 2005

[11] Mitrovic, A., Martin, B., Suraweera, P. Constraint-based tutors: past, present and future. IEEE Intelligent Systems, special issue on Intelligent Educational Systems, vol. 22, no. 4, pp. 38-45, July/August 2007.

[12] Suraweera, P., Mitrovic, A. An Intelligent Tutoring System for Entity Relationship Modeling Int. J. Artificial Intelligence in Education, vol. 14, no. 3-4, 375-417, 2004.

[13] Hartley, D., Mitrovic, A. Supporting learning by opening the student model. In: S. Cerri, G. Gouarderes and F. Paraguacu (eds.) Proc. 6th

Int. Conf on Intelligent Tutoring Systems ITS 2002, Biarritz, France, LCNS 2363, 453-462, 2002.

[14] J.M.P. van Vaveren. The Quake III Arena Bot. Master's thesis, TU Delft, June 2001.

[15] Kobsa, A.: Generic User Modeling Systems, User Modeling and User-Adapted Interaction, v.11 n.1-2, p.49-63, 2001.

[16] Cifaldi, F.: Analysts: FPS 'Most Attractive' Genre for Publishers, GamaSutra, February 21, 2006

[17] VanLehn, K.: The behavior of tutoring systems. International Journal of Artificial Intelligence in Education. 16, 3, p. 227-265, 2006

[18] Corbett, A. T., Koedinger, K. R., & Anderson, J. R.: Intelligent tutoring systems. In Helander, M. G., Landauer, T. K., & Prabhu, P. V. (Ed.s) Handbook of Human-Computer Interaction, p. 849-874, 1997

[19] Smyth, B., Coyle, M., Briggs, P.: Google? Shared! Adding Social Search Technologies to Mainstream Search Engines. User Modeling, Apadtation and Personalization 2009.

[20] Woolf, B.: Building Intelligent Interactive Tutors, Morgan Kauffman, New York, NY, 2008.

[21] Self, J. Bypassing the intractable problem of student modelling. In Proceedings of the Intelligent Tutoring Systems conference, ITS'88, pages 18-24, 1988.

[22] Jameson, A.: Adaptive Interfaces and Agents. In: A. Sears & J. A. Jacko (Eds.), The human-computer interaction handbook: Fundamentals, evolving technologies and emerging applications (2nd ed.) p. 433-458. Boca Raton, FL: CRC Press, 2008

[23] Miles, J.: Machine learning for adaptive computer game opponents. MSc Thesis, The University of Waikato, 2009.